Oracle® Communications Billing and Revenue Management

Developer's Reference

Release 15.0
F86227-02
June 2024

Oracle Communications Billing and Revenue Management Developer's Reference, Release 15.0

F86227-02

Contents

4 Storable Class-to-SQL Mapping

5 Sample Applications

Preface

This guide provides reference information for Oracle Communications Billing and Revenue Management (BRM) application programming interfaces (APIs).

This guide has been updated to include changes and new feature content added for release 15.0.1.

Audience

This guide is intended for developers.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1

PIN Libraries Reference

This chapter provides reference information for Oracle Communications Billing and Revenue Management (BRM) Portal Information Network (PIN) libraries.

Configuration File-Reading Functions

Use these functions to read configuration files, such as **pin.conf** files.

pin_conf

This BRM library routine reads a single configuration value from a configuration file.

The Connection Manager (CM), Data Manager (DM), and Portal Communications Module (PCM) libraries all use this routine to read the configuration information.

When first called, this routine looks for the configuration file specific to the application. See "Locations of Configuration and Properties Files" in *BRM System Administrator's Guide*. The library returns an error if it cannot locate the configuration file.

This routine uses regular **malloc**. If you are using this routine in a Storage Manager to get data to put on an flist, use **SET** (*not* **PUT**), and then free the object by using the regular **free** routine when you are finished.

> **Note:**
>
> Do not use this routine if performance is a consideration and you use the routine often.

For more information on configuration files, see "Syntax for Configuration Entries" in *BRM System Administrator's Guide*.

For information on reading multiple configuration values from a file, see "pin_conf_multi".

Syntax

```
#include "pcm.h"
void
pin_conf(
                char          *prog_name,
                char          *token,
                int32          valtype,
                caddr_t*     **valpp,
                int32         *errp);
```

Parameters

prog_name
The program name this routine looks for in the configuration file. If *prog_name* is **NULL**, the routine looks only for entries marked with a program of "-". If *prog_name* is any other value, the routine looks for either a specific match or "-" in the program parameter.

token
The name of the configuration entry keyword this routine looks for in the configuration file.

valtype
The **type** of the value the routine reads in the configuration entry. This parameter tells the routine how to interpret the entry value. The supported types are:

- **PIN_FLDT_INT**
- **PIN_FLDT_DECIMAL**
- **PIN_FLDT_STR**
- **PIN_FLDT_POID**

valpp
The **ptr-ptr** used to pass back the location of the value for the entry. The memory for the value is dynamically allocated, and the filled-in pointer **type** matches the value **type**.

errp
A pointer to the error buffer, which passes error information back to the caller.

Return Values

This routine returns nothing.

This routine passes error status back to the caller. If it finds a matching entry in the configuration file, it passes back **PIN_ERR_NONE**. If it does not find a matching entry, it passes back **PIN_ERR_NOT_FOUND**. The routine might also pass back other error values.

pin_conf_beid

This library routine reads values for BRM balance elements from the **/config/beid** object.

Syntax

```
#include "pin_errs.h"
#include "pcm.h"
pin_flist_t*
pin_conf_beid(
               pcm_context_t     *ctxp,
               pin_errbuf_t      *ebufp);
```

Parameters

ctxp
A pointer to an open context. This routine gets the database number from the configuration file of the current application and queries that database for the **/config/beid** object.

ebufp
A pointer to the error buffer, which passes error information back to the caller.

Return Values

Returns values for the **/config/beid** object data as an flist.

Error Handling

This routine sets the return flist to **NULL** and provides more information about the error in the error buffer if there is an error.

pin_conf_multi

This library routine reads multiple configuration values of the same type from a configuration file. To do this, you reuse this routine until it returns **PIN_ERR_NOT_FOUND**. This routine uses the **time_t** value to monitor the configuration file for changes throughout this operation and returns an error if the state of the file changes.

The Connection Manager (CM), Data Manager (DM), and PCM libraries all use this routine to read the configuration information.

When first called, this routine looks for the configuration file specific to the application. See "Configuration File Locations" in *BRM System Administrator's Guide*. The library returns an error if it cannot locate the configuration file.

This routine uses regular **malloc**. If you are using this routine in a Storage Manager to get data to put on an flist, use **SET** (*not* **PUT**), and then free the object by using the regular **free** routine when you are finished.

> **Note:**
>
> Do not use this routine if performance is a consideration and you use the routine often.

For more information on configuration files, see "Using Configuration Files to Connect and Configure Components" in *BRM System Administrator's Guide*.

For information on reading a single configuration value from a file, see "pin_conf".

Syntax

```
#include "pcm.h"
void
pin_conf(
                char          *prog_name,
                char          *token,
                int32          valtype,
                caddr_t*     **valpp,
                int32         *linep,
                time_t        *modtp,
                int32         *errp);
```

Parameters

prog_name
The program name this routine looks for in the configuration file. If *prog_name* is **NULL**, the routine looks only for entries marked with a program of "-". If *prog_name* is any other value, the routine looks for either a specific match or "-" in the program parameter. For a description of configuration file syntax, see "Syntax for Configuration Entries" in *BRM System Administrator's Guide*.

token
The name of the configuration entry keyword this routine looks for in the configuration file.

valtype
The **type** of the value the routine reads in the configuration entry. This parameter tells the routine how to interpret the entry value. The supported types are:

- **PIN_FLDT_INT**
- **PIN_FLDT_DECIMAL**
- **PIN_FLDT_STR**
- **PIN_FLDT_POID**

valpp
The **ptr-ptr** used to pass back the location of the value for the entry. The memory for the value is dynamically allocated, and the filled-in pointer **type** matches the value **type**.

linep
A pointer to a line number. Passes an integer back to the caller to identify the line where the last value was found. Initialize to zero on the first call.

modtp
A pointer to a time variable. Passes a timestamp back to the caller to compare to the last timestamp. Initialize to zero on the first call.

errp
A pointer to the error status, which passes error information back to the caller.

Return Values

This routine returns nothing.

This routine passes error status back to the caller.

- If it finds a matching entry in the configuration file, it passes back **PIN_ERR_NONE**. This indicates that the routine then reuses the key to look for another matching entry (as long as it has not generated a PIN_ERR_STALE_CONF error).
- If it does not find a matching entry, it passes back **PIN_ERR_NOT_FOUND**. This signals the end of the routine.
- If it detects, based on a change in the **time_t** value, that the configuration file has been opened, modified, or has otherwise changed since it first accessed the file (jeopardizing the ability of the routine to maintain correct reference to the last value read), it passes back **PIN_ERR_STALE_CONF**.

> **Note:**
>
> In this case, you must restart the entire process.

The routine may also pass back other error values.

Decimal Data Type Manipulation Functions

This section describes decimal data type manipulation functions.

About Using the API

The decimal data type application programming interface (API) consists of a minimal set of methods that provides all the functionality you need to perform basic mathematical functions, comparison, and format conversion with the decimal data type. Input and output to the functions are provided using number strings or floating point doubles.

> **Note:**
>
> Use strings to avoid small quantity errors; for example, 31.299999999 vs. 31.3.

If there are errors, functions that return a **pin_decimal_t** return **NULL**. **pbo_decimal_destroy** allows **NULL**.

International Platform Issues

The **pin_decimal** function expects the decimal point character to be that of the locale. For US systems, this is a period; for most international platforms, it is a comma.

> **Caution:**
>
> Do not pass a string with a hard-coded decimal point to **::pin_decimal** because **pin_decimal** will return a NULL pointer in platforms that do not use a period for the decimal point character.

About Rounding Modes

This section defines the rounding modes that you pass as input parameters in the following functions:

- pbo_decimal_round
- pbo_decimal_round_assign
- pbo_decimal_from_double
- pbo_decimal_from_double_round

The rounding modes in Table 1-1 are defined in **pcm.h**. They have the same names and functionality as the Java BigDecimal Datatype.

Table 1-1 Rounding Modes

Rounding Mode	Description
ROUND_UP	Rounds up to the nearest number of the appropriate scale. **Examples**: 21.11 rounds to 21.2 when the scale is one decimal place.
ROUND_DOWN	Rounds down to the nearest number of the appropriate scale. **Examples**: 21.19 rounds to 21.1 when the scale is one decimal place.
ROUND_DOWN_ALT	Rounds down after first rounding to the nearest using a scale of two more than the one configured. This method compensates for possible loss of precision when numbers are rounded down during certain computations, such as when prorating cycle fees. For more information, see "About Rounding Modes That Correct for Loss of Precision" in *BRM PDC Creating Product Offerings*.
ROUND_CEILING	If the number is positive, rounding is the same as for ROUND_UP; if negative, the same as for ROUND_DOWN.
ROUND_FLOOR	If the number is positive, rounding is the same as for ROUND_DOWN; if negative the same as for ROUND_UP. This method allows you to round to benefit customers. For example, if rounding is set to two significant digits, a credit to a customer of -7.999 is rounded to -8.00, and a debit of 7.999 is rounded to 7.99.
ROUND_FLOOR_ALT	Rounds using ROUND_FLOOR after first rounding to the nearest using a scale of two more than the one configured. This method compensates for possible loss of precision when numbers are rounded down during certain computations, such as when prorating cycle fees. For more information, see "About Rounding Modes That Correct for Loss of Precision" in *BRM PDC Creating Product Offerings*.
ROUND_HALF_UP	If the discard part is .5 or higher round up; otherwise, round down. **Examples**: 21.15 rounds to 21.2, 21.14 rounds to 21.1, etc. This is the most common rounding method.
ROUND_HALF_DOWN	If the discard part is more than .5, round up; if it is .5 or less, round down. **Examples**: 21.16 rounds to 21.2, 21.15 rounds to 21.1.
ROUND_HALF_EVEN	If the digit to the left of the discard is odd, rounding is the same as for ROUND_HALF_UP. If the digit to the left is even, rounding is the same as for ROUND_HALF_DOWN. **Examples:** 1.049 rounds to 1.0 1.050 rounds to 1.0 1.051 rounds to 1.1 1.149 rounds to 1.1 1.150 rounds to 1.2 1.151 rounds to 1.2
ROUND_UNNECESSARY	Rounding not allowed. If rounding is attempted with this rounding mode, an error is returned.

About Scaling

A decimal data type is based on the Java BigDecimal data type. It is an immutable, arbitrary-precision signed decimal number, which consists of an arbitrary precision integer value and a nonnegative integer scale, which represents the number of decimal digits to the right of the decimal point.

For this implementation, the scale is set at 15, meaning numbers carry up to 15 decimal places. For operations that would normally result in a value with a larger scale, the value is rounded to 15 decimal places. For example, when multiplying the two decimal data types 12.528694120521357 and 4.126943650923412, the mathematical result would normally be 51.705214655047095455751917310084, which has a scale of 30. However, because the scale is set at 15, the product is rounded to 51.705214655047095 and a consistent scale of 15 is maintained.

About Memory Management

For functions that allocate memory for the **pin_decimal_t** structure, make sure that the memory is reclaimed after the **pin_decimal_t** is no longer needed. If **pin_decimal_t** has been passed to an flist with PIN_FLIST_PUT, use **pin_flist_destroy** to reclaim memory. Otherwise, use **pbo_decimal_destroy**.

assign functions do not allocate new memory; instead, they replace the first parameter with the new value. Therefore, there is no need to reclaim memory.

pbo_decimal_abs

This function returns a pointer to a newly allocated **pin_decimal_t**, which is the absolute value of the input **pin_decimal_t**.

Syntax

```
pin_decimal_t*
pbo_decimal_abs(
                const pin_decimal_t     *pdp,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp
A pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_abs_assign

This function replaces the input **pin_decimal_t** with its absolute value.

Syntax

```
pin_decimal_t*
pbo_decimal_abs_assign(
               pin_decimal_t     *pdp,
               pin_errbuf_t      *ebufp);
```

Parameters

pdp
A pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_add

This function adds the two decimals passed in and returns a pointer to a newly allocated **pin_decimal_t**. The scale of the output is the larger of the scales of the two inputs.

Syntax

```
pin_decimal_t*
pbo_decimal_add(
               const pin_decimal_t     *pdp1,
               const pin_decimal_t     *pdp2,
               pin_errbuf_t            *ebufp);
```

Parameters

pdp1
A pointer to the input **pin_decimal_t**.

pdp2
A pointer to another input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**

- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_add_assign

This function replaces the value of the first **pin_decimal_t** with the sum of itself and another **pin_decimal_t**.

Syntax

```
void
pbo_decimal_add_assign(
                pin_decimal_t           *pdp1,
                const pin_decimal_t     *pdp2,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp
A pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_compare

This function compares the first input decimal with the second input decimal and returns one of the following values to indicate the difference between the input decimals:

- **-1** if *pdp1* < *pdp2*
- **0** if *pdp1* = *pdp2*

> **Note:**
>
> *pdp1* is considered equal to *pdp2* if the difference between them is less than 10^{-12}.

- **1** if *pdp1* > *pdp2*
- **0** in the event of an error.

Syntax

```
int
pbo_decimal_compare(
                const pin_decimal_t     *pdp1,
```

```
                const pin_decimal_t     *pdp2,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp1
A pointer to the first **pin_decimal_t**.

pdp2
A pointer to the second **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_copy

This function makes a copy of the input **pin_decimal_t** and returns a pointer to the newly allocated **pin_decimal_t**.

Syntax

```
pin_decimal_t*
pbo_decimal_copy(
                const pin_decimal_t     *pdp,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp
A pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_destroy

This function frees all the memory associated with the specified **pin_decimal_t** and sets **decpp* to NULL.

Syntax

```
void
pbo_decimal_destroy(
                pin_decimal_t     **decpp);
```

Parameter

decpp
A pointer to a pointer to the **pin_decimal_t** to be deleted. Can be set to **NULL** (the function does nothing).

pbo_decimal_divide

This function divides the first input parameter by the second input parameter and returns a pointer to a newly allocated **pin_decimal_t**.

> **Note:**
>
> Rounding is performed according to preset rounding and scaling. The default rounding mode is ROUND_DOWN and the scaling is set at 15 decimal places.

Syntax

```
pin_decimal_t*
pbo_decimal_divide(
                const pin_decimal_t     *nump,
                const pin_decimal_t     *byp,
                pin_errbuf_t            *ebufp);
```

Parameters

nump
A pointer to the dividend.

byp
A pointer to the divisor.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_BAD_ARG if one of the following is true:
  - The scale is less than 0.
  - The rounding mode is unknown.
  - Either the dividend or the divisor is not a valid **pin_decimal_t**.
  - An attempt was made to divide by 0.

- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_divide_assign

This function divides the dividend by the divisor and stores the result in the dividend.

Syntax

```
void
pbo_decimal_divide_assign(
                pin_decimal_t           *nump,
                const pin_decimal_t     *byp,
                pin_errbuf_t            *ebufp);
```

Parameters

nump
A pointer to the dividend.

byp
A pointer to the divisor.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_BAD_ARG if one of the following is true:
  - The scale is less than 0.
  - The rounding mode is unknown.
  - Either the dividend or the divisor is not a valid **pin_decimal_t**.
  - An attempt was made to divide by 0.
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_from_double

This function constructs a **pin_decimal_t** data type from the double-precision floating point number (allocates memory) and returns a pointer to the newly created **pin_decimal_t** data type.

> **Note:**
>
> Because of the inherent rounding errors associated with converting a double to a decimal data type, you should avoid using this function whenever possible. Use **pbo_decimal_from_str** instead. If you must use doubles, use the **pbo_decimal_from_double_round** function.

Syntax

```
pin_decimal_t
*pbo_decimal_from_double(
               double            d,
               pin_errbuf_t     *ebufp);
```

Parameters

d
The input of type double float (a double-precision floating point number).

ebufp
A pointer to the error buffer.
See also "pbo_decimal_from_str ".

pbo_decimal_from_double_round

This function provides an option for choosing the rounding mode. (See "About Rounding Modes".)

Constructs a **pin_decimal_t** data type from the double-precision floating point number (allocates memory) and returns a pointer to the newly created **pin_decimal_t** data type.

> **Note:**
>
> Because of the inherent rounding errors associated with converting a double to a decimal data type, you should avoid using this function whenever possible. Use **pbo_decimal_from_str** instead.

Syntax

```
pin_decimal_t*
pbo_decimal_from_double_round(
               double            value,
               int               rounding_mode,
               pin_errbuf_t     *ebufp)
```

Parameters

value
The value to convert.

rounding_mode
See "About Rounding Modes".

ebufp
A pointer to the error buffer.

pbo_decimal_from_str

This function constructs a **pin_decimal_t** data type from an input string and returns a pointer to the newly created **pin_decimal_t** data type.

This function understands **NULL** to create a NULL-valued **pin_decimal_t**. The string does not need to end with a null character, but parsing will end at either a null character or any white space character.

This function ignores leading spaces, tabs, and leading 0's and checks on nonnumeric types.

This function detects the sign (+ or -) and stores it. This function accepts the same input at **strtod** except that an exponent is not allowed, and only base 10 is supported.

Syntax

```
pin_decimal_t*
pbo_decimal_from_str(
               const            *str,
               pin_errbuf_t     *ebufp);
```

Parameters

str
The input number string.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the string pointer is **NULL**
- PIN_ERR_BAD_ARG if there were multiple decimal points before null or space or if it cannot derive a valid number from the string
- PIN_ERR_NO_MEM if the function cannot allocate memory for **pbo_decimal**

pbo_decimal_is_null

This function verifies if the input **pin_decimal_t** is **NULL**.

Syntax

```
int
pbo_decimal_is_null(
               const pin_decimal_t     *pdp,
               pin_errbuf_t            *ebufp);
```

Parameters

pdp
The pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns PIN_ERR_BAD_ARG indicating that a non-NULL pointer points to a data area not marked as a valid **pin_decimal_t**.

pbo_decimal_is_zero

This function checks if the input value is a valid **pin_decimal_t** and has a zero value. Returns **1** if the conditions are met; otherwise, it returns **0**.

Syntax

```
int
pbo_decimal_is_zero(
                const pin_decimal_t     *pdp,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp
A pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns PIN_ERR_BAD_ARG indicating that a non-NULL pointer points to a data area that is not marked as a valid **pin_decimal_t**.

pbo_decimal_multiply

This function multiplies the two input **pin_decimal_t** values and returns a pointer to a new **pin_decimal_t** that is the product.

Syntax

```
pin_decimal_t*
pbo_decimal_multiply(
                const pin_decimal_t     *pdp1,
                const pin_decimal_t     *pdp2,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp1
The pointer to an input **pin_decimal_t**.

pdp2
The pointer to another input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_multiply_assign

This function multiplies two **pin_decimal_t** data types and stores the product in the first **pin_decimal_t**.

For example, if **a=**10 and **b=**2, after calling **pbo_decimal_multiply_assign(a, b, *ebufp)**, **a** is equal to 20.

Syntax

```
void
pbo_decimal_multiply_assign(
                pin_decimal_t           *pdp1,
                const pin_decimal_t     *pdp2,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp1
The pointer to an input **pin_decimal_t**.

pdp2
The pointer to another input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued

pbo_decimal_negate

This function returns a pointer to a new **pin_decimal_t** that has the reverse sign of the input decimal. If the input decimal has a value of **0**, it returns a pointer to another **pin_decimal_t** with the value of **0**.

Table 1-2 contains examples, where *x* is a pointer **pin_decimal_t**:

Table 1-2 pbo_decimal_negate Examples

Value to Which x Points	pbo_decimal_negate(x, ebuf) Returns a New Pointer to This Value:
5	-5
0	0
-3	3

Syntax

```
pin_decimal_t*
pbo_decimal_negate(
                const pin_decimal_t     *pdp,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp
The pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_negate_assign

This function reverses the sign of the input **pin_decimal_t**.

Syntax

```
pin_decimal_t*
pbo_decimal_negate_assign(
               pin_decimal_t     *pdp,
               pin_errbuf_t      *ebufp);
```

Parameters

pdp
The pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued

pbo_decimal_round

This function returns a pointer to a new **pin_decimal_t** that contains the value of the first argument rounded according to the specified scale and rounding mode.

Syntax

```
pin_decimal_t*
pbo_decimal_round(
               const pin_decimal_t     *decp,
               int32                    scale,
               int32                    rounding_mode,
               pin_errbuf_t            *ebufp);
```

Parameters

decp
A pointer to the input **pin_decimal_t**.

scale
See "About Scaling".

rounding_mode
See "About Rounding Modes".

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_round_assign

This function replaces the value of the first argument with the value of the argument rounded according to the specified scale and rounding mode.

Syntax

```
void
pbo_decimal_round_assign(
                pin_decimal_t     *decp,
                int32              scale,
                int32              rounding_mode,
                pin_errbuf_t      *ebufp);
```

Parameters

decp
A pointer to the input **pin_decimal_t**.

scale
See "About Scaling".

rounding_mode
See "About Rounding Modes".

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is NULL
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued

- PIN_ERR_BAD_ARG if *decp* is an invalid value

pbo_decimal_sign

This function returns the sign of the **pin_decimal_t** argument: **-1** if the argument is negative, **0** if the argument is zero or if there is an error, or **1** if the argument is positive.

Syntax

```
int
pbo_decimal_sign(
                const pin_decimal_t     *pdp,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp
The pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued

pbo_decimal_subtract

This function subtracts two **pin_decimal_t** parameters and returns a pointer to a new **pin_decimal_t** containing the difference.

Syntax

```
pin_decimal_t*
pbo_decimal_subtract(
                const pin_decimal_t     *nump,
                const pin_decimal_t     *byp,
                pin_errbuf_t            *ebufp);
```

Parameters

nump
The pointer to the **pin_decimal_t** from which to subtract.

byp
The pointer to the **pin_decimal_t** to subtract.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**

- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

pbo_decimal_subtract_assign

This function subtracts a decimal from another decimal and replaces the value of the first decimal with the difference.

For example, if **a=**8 and **b=**3, after calling **pbo_decimal_subtract_assign (a, b, ebuf)**, **a** is equal to **5**.

Syntax

```
void
pbo_decimal_subtract_assign(
                pin_decimal_t           *pdp1,
                const pin_decimal_t     *pdp2,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp1
The pointer to an input **pin_decimal_t**.

pdp2
The pointer to another input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued

pbo_decimal_to_double

This function converts the input **pin_decimal_t** into a double-precision floating point number.

If **pin_decimal_t** is not **NULL**, this function converts **pin_decimal_t** to a string using **pin_decimal_to_str(NULL format,...)** and then **strtod**.

Syntax

```
double
pbo_decimal_to_double(
                const pin_decimal_t     *pdp,
                pin_errbuf_t            *ebufp);
```

Parameters

pdp
A pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**
- PIN_ERR_BAD_ARG if **strtod** returns an error

See also **pin_decimal_to_str().**

pbo_decimal_to_str

This function creates an ASCII string representation of the input decimal value.

If successful, the function returns a pointer to the allocated null-terminated string. If there are errors, it returns **NULL**.

Syntax

```
char*
pbo_decimal_to_str(
               const pin_decimal_t     *pdp,
               pin_errbuf_t            *ebufp);
```

Parameters

pdp
A pointer to the input **pin_decimal_t**.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pin_decimal_t** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **pin_decimal_t** is **NULL**-valued
- PIN_ERR_NO_MEM if the function cannot allocate memory for the output **pin_decimal_t**

psiu_currency_append_currency_exchange_rate

The function appends a **psiu_currency_exchange_rate**_t structure to the designated rate table.

Syntax

```
void
psiu_currency_exchange_rate (
        psiu_list_t                     *psiu_plist_cerp,
```

```
        psiu_currency_exchange_rate_t    *pcerRatep,
        pin_errbuf_t                     *ebufp);
```

Parameters

psiu_plist_cerp
The rate table to append the **psiu_currency_exchange_rate_t** structure into.

pcerRatep
A pointer to the **psiu_currency_exchange_rate_t** structure that should be appended to the specified list.

ebufp
A pointer to the error buffer.

Error Handling

If there are errors, this function returns the following error status:

- PIN_ERR_NULL_PTR if the input **pcerRatep** pointer is **NULL**
- PIN_ERR_IS_NULL if the input **psiu_plist_cerp** is **NULL**-valued

Error-Handling Macros

This section describes error-handling macros.

PIN_ERR_LOG_EBUF

This BRM macro logs a standardized message that includes details of the error condition recorded in an error buffer. It provides a convenient method for logging errors returned by API calls that use the error buffer to pass back status. The caller can specify an additional message that is appended to the standard format.

Syntax

```
#include "pcm.h"
void
PIN_ERR_LOG_EBUF(
                int32          level,
                char          *msg,
                pin_errbuf_t  *ebufp);
```

Parameters

level
The level of this log message. Based on the level specified and the logging level set in the log system, the message is either printed or discarded. See "PIN_ERR_SET_LEVEL" for the error level descriptions.

msg
A string to be printed in addition to the standard logging message. Allows additional detailed information to be added to the log message by the caller.

ebufp
A pointer to the error buffer containing the error condition. The values in the error buffer are printed in human-readable form as part of the log message.

Return Values

This macro returns nothing.

Error Handling

There are no error conditions for this macro. If the message cannot be logged for any reason, that information is not passed back to the caller.

PIN_ERR_LOG_FLIST

This macro prints the contents of an flist to the error log file. It allows an application to log an arbitrary message and the corresponding flist for recording errors, accounting, or debugging. The specified message and flist are logged in the standard log entry format, so complete information about where they came from is available in the log file.

Syntax

```
#include "pcm.h"
void
PIN_ERR_LOG_FLIST(
                int32         level,
                char         *msg,
                pin_flist_t  *flistp);
```

Parameters

level
The level of this log message. Based on the level specified and the logging level set in the log system, the message is either printed or discarded. See "PIN_ERR_SET_LEVEL" for the error-level descriptions.

msg
A string to be printed in addition to the standard logging message. Allows additional detailed information to be added to the log message by the caller.

flistp
A pointer to the flist to be printed in addition to the log message.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_ERR_LOG_MSG

This macro logs the specified message to the log file. It allows an application to log arbitrary messages for recording errors or debug information. The specified message is logged in the standard log entry format, so complete information about where the message came from is available in the log file.

Syntax

```
#include "pcm.h"
void
PIN_ERR_LOG_MSG(
                int32     level,
                char     *msg);
```

Parameters

level
The level of this log message. Based on the level specified and the logging level set in the log system, the message is either printed or discarded. See "PIN_ERR_SET_LEVEL" for the error-level descriptions.

msg
A string to be printed in addition to the standard logging message. Allows additional detailed information to be added to the log message by the caller. Special characters should be escaped if you want them to be printed without modification.

Return Values

This macro returns nothing.

Error Handling

There are no error conditions for this macro. If the message cannot be logged for any reason, that information is not passed back to the caller.

PIN_ERR_LOG_POID

This macro prints the contents of a POID to the error log file. This operation allows an application to log an arbitrary message and the corresponding POID for recording errors, accounting, or debugging. The specified message and POID are logged in the standard log entry format, so complete information about where they came from is available in the log file.

Syntax

```
#include "pcm.h"
void
PIN_ERR_LOG_POID(
                int32     level,
                char     *msg,
                poid_t   *pdp);
```

Parameters

level
The level of this log message. Based on the level specified and the logging level set in the log system, the message is either printed or discarded. See "PIN_ERR_SET_LEVEL" for the error-level descriptions.

msg
A string to be printed in addition to the standard logging message. Allows additional detailed information to be added to the log message by the caller.

pdp
A pointer to the POID to be printed in addition to the standard log entry information.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_ERR_SET_LEVEL

This macro sets the desired level of logging. Messages sent to the logging system have a severity code that describes the category of the message. Users can choose to have messages of different categories either logged or suppressed, depending on how much logging output they would like to see. Messages that are suppressed are discarded.

In general, BRM recommends that only debug messages be suppressed on a production system. All other types of messages convey possible system problems that should be investigated. Debug messages can be enabled when they might help diagnose an application error and then suppressed when the system is running in a steady state.

If PIN_ERR_SET_LEVEL is not called, the logging system defaults to a level of 2.

Syntax

```
#include "pcm.h"
int32
PIN_ERR_SET_LEVEL(
                int32     level);
```

Parameter

level
Sets the mask for which level of errors should be logged and which ones suppressed. All messages with a level of *level* or less are printed. All messages with a level greater than *level* are suppressed. Errors come in the levels listed in Table 1-3:

Table 1-3 PIN_ERR_SET_LEVEL Values

Allowed Level Values	System Category	Type of Message	Messages Returned
0	N/A	N/A	Nothing at this level
1	**E**	Error	Serious system integrity problems
2	**W**	Warning	Possible data corruption problems
3	**D**	Debug	Details of application errors

- Setting *level* to **0** means no messages will be produced, no matter what the error.
- Setting *level* to **1** will log only errors, which indicate some portion of the BRM system is not operating correctly.
- Setting *level* to **2** will print errors and warnings. Warnings indicate that data was found in the database that is suspect, and some data corruption may have occurred. The system can still operate properly, but specific operations related to the corrupt data may have to be bypassed.
- Setting *level* to **3** prints debug messages. The debug messages log detailed information about operations that applications attempt that generate errors in the system due to incorrect parameters or other application level errors. The system is not adversely affected by this type of event, but the application developer can use the debug messages to more easily pinpoint where the application error is located.

Return Values

Returns **0** if the macro is successful. Returns a nonzero value if an error occurred. The only possible failure is the specification of an unreasonable value for *level*.

Error Handling

Returns a nonzero value if an error occurred. In this case, the internal state of the logging system is unchanged.

PIN_ERR_SET_LOGFILE

This macro specifies the file to use for logging. The log file can be changed at any time by calling PIN_ERR_SET_LOGFILE. All messages logged after the change are logged to the new file.

If this macro is not called, the logging system uses the default **./default.pinlog** log file, where **./** is relative to the directory in which the application was started.

Syntax

```
#include "pcm.h"
int32
PIN_ERR_SET_LOGFILE(
                char     *path);
```

Parameter

path
The path of the file to be used as the log file. The file is opened exactly as specified, so relative paths will work, but they will be relative to the current directory of the running program.

Return Values

Returns a nonzero value if an error occurred.

Error Handling

Returns a nonzero value if an error occurred. The internal state of the logging system is unchanged. The return value should be tested after the call to ensure the desired log file will be used.

PIN_ERR_SET_PROGRAM

This macro sets the program name for log messages. The program name is printed in each log message as additional information to aid in debugging problems. The program name can be set to any string desired.

If PIN_ERR_SET_PROGRAM is not called, log messages are printed with a blank program name field.

Syntax

```
#include "pcm.h"
int32
PIN_ERR_SET_PROGRAM(
                char     *program);
```

Parameter

program
The name of the running program to be printed in log messages. If the pointer is NULL, the current name is not changed.

Return Values

Returns **0** if the macro is successful. Returns a nonzero value if an error occurred. The only possible failure condition is the specification of a NULL pointer.

Error Handling

Returns a nonzero return value if an error occurred. In this case, the internal state of the logging system is unchanged.

PIN_ERRBUF_CLEAR

This macro is used for a newly allocated or defined error buffer structure to initialize the contents of the error buffer to **0**.

Syntax

```
#include "pcm.h"
void
PIN_ERRBUF_CLEAR(
                pin_errbuf_t     *ebufp);
```

Parameter

ebufp
A pointer to the error buffer that is initialized.

Return Values

This macro returns nothing.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home*/**source/samples/app/c**.

PIN_ERRBUF_IS_ERR

This macro checks the specified error buffer for an error condition. It allows an application to quickly check whether an error occurs on a call that uses the error buffer.

Macros that use individual ebuf error handling must use PIN_ERRBUF_IS_ERR after each call to test for an error.

Macros that use series-style ebuf error handling can make an entire series of calls and use this macro once at the end to test for an error.

Syntax

```
#include "pcm.h"
int32
PIN_ERRBUF_IS_ERR(
                pin_errbuf_t     *ebufp);
```

Parameter

ebufp
A pointer to an error buffer. Used by the macro to determine whether an error has occurred.

Return Values

Returns **0** if the error buffer contains no error. Returns a nonzero value if the error buffer contains an error.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home*/**source/samples/app/c**.

PIN_ERRBUF_RESET

This macro is called to reset the error buffer either before reusing an existing error buffer structure or before calling **pin_free** to free a dynamically allocated error buffer structure.

For details on the structure and fields in an error buffer, see "Error Buffer" in *BRM Developer's Guide*.

The use of PIN_ERRBUF_RESET depends on the type of macro called with the error buffer:

- **Individual-style ebuf:** Macros that use this style of error handling must examine the error buffer for an error after each call. Use PIN_ERRBUF_RESET to clear any error that was detected before using the same error buffer again.
- **Series-style ebuf:** Macros that use this style of error handling can use the same error buffer for a series of calls without checking for or clearing errors between calls. After a series of calls, check the error buffer for errors. Use PIN_ERRBUF_RESET to clear any error before using the error buffer again.

Syntax

```
#include "pcm.h"
void
PIN_ERRBUF_RESET(
                pin_errbuf_t     *ebufp);
```

Parameter

ebufp
A pointer to the error buffer that is reset.

Return Values

This macro returns nothing.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

pin_set_err

This function sets the error values in the **pin_errbuf_t (ebuf)** structure pointer.

> **Note:**
>
> This is the only error handling routine that is not a macro. This is a function.

Syntax

```
EXTERN
void
pin_set_err(
               pin_errbuf_t    *ebuf,
               int32            location,
               int32            pin_errclass,
               int32            pin_err,
               int32            field,
               int32            rec_ID,
               int32            reserved);
```

Parameters

ebuf
A pointer to the error buffer.

location
The location of an error. For a list of possible locations, see "BRM Error Locations" in *BRM System Administrator's Guide*.

pin_errclass
One of the four classes. See "BRM Error Classes" in *BRM System Administrator's Guide*.

pin_err
One of the system error codes. For a list of possible error codes, see "BRM Error Codes" in *BRM System Administrator's Guide*.

field
Set to **0** or to the applicable **PIN_FLD**_*xxx*.

rec_ID
Set to **0** or to the record ID of the array element the error occurred on.

reserved
Set to **0** or to a value chosen to provide further information about the specific error.

Return Values

This function returns nothing.

Error Handling

There are no error conditions for this function. If the message cannot be logged for any reason, that information is not passed back to the caller.

Flist Field-Handling Macros

This section describes flist field-handling macros.

PIN_FLIST_ANY_GET_NEXT

This BRM macro gets the value of the next simple field, substructure, or element of an array in an flist. It lets an application walk an flist retrieving each field value.

The value returned is a pointer to the actual field value, and the field remains unchanged on the original flist. The value returned must be treated as read-only to maintain the integrity of the flist. If a writable copy of the value is needed, the application must either make a copy of the returned value or take it according to its type as listed in Table 1-4:

Table 1-4 Next Field Macros

Field Type	Macro to Use
Simple	PIN_FLIST_FLD_TAKE
Substructure	PIN_FLIST_SUBSTR_TAKE
Array element	PIN_FLIST_ELEM_TAKE

Syntax

```
#include "pcm.h"
void
*PIN_FLIST_ANY_GET_NEXT(
                pin_flist_t     *flistp,
                pin_fld_num_t   *fldp,
                int32           *record_idp,
                pin_cookie_t    *cookiep,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist containing the field being obtained.

fldp
A pointer to the field.

record_idp
The element ID, in case of array field is returned if not **NULL**.

cookiep
The cookie for the next field.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the value on the flist. The pointer must be cast appropriately depending on the type of the field. Returns **NULL** if an error occurred or if the field is not found.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_ADD

This macro adds a specified array element to the flist. The flist for the element fields is created and returned. The pointer to this element flist can then be used to set/put fields into the element.

If the specified array element already exists on the flist, the existing element flist is destroyed and replaced by the new element flist.

Syntax

```
#include "pcm.h"
pin_flist_t *
PIN_FLIST_ELEM_ADD(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                v_int32          elem_id,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist receiving the array element.

fld
The number of the field being added.

elem_id
The element ID of the element being added.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the flist for the array element. Returns **NULL** if an error occurred.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_FLIST_ELEM_COPY

This macro copies an element in an array from one flist to another. You can change the element name and record ID while copying the element. The type must remain the same.

Syntax

```
#include "pcm.h"
int32
PIN_FLIST_ELEM_COPY(
                pin_flist_t    *src_flistp,
                pin_fld_num_t   src_fld,
                pin_rec_id_t    src_recID,
                pin_flist_t    *dest_flistp,
                pin_fld_num_t   dest_fld,
                pin_rec_id_t    dest_recID,
                pin_errbuf_t    *ebufp );
```

Parameters

src_flistp
A pointer to the source flist from which the element is copied.

src_fld
The element that is copied from the source flist.

src_recID
The record ID of the element that is copied.

dest_flistp
A pointer to the destination flist to which an element is copied.

dest_fld
The copied element in the destination flist.

dest_recID
The record ID of the copied element in the destination flist.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns **1** if the field to be copied is found. Returns **0** if the field to be copied is not found. Not finding a field does not result in an error buffer error.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_COUNT

This macro counts the number of elements of an array on an flist. It does not look at substructure flists, so the elements must be on the flist passed in at the highest level.

Syntax

```
#include "pcm.h"
int32
PIN_FLIST_ELEM_COUNT(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist being counted.

fld
The field number of the array containing the elements being counted. Each time a field with this number is found, the element count is incremented.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns the number of elements found as an unsigned integer. Returns **0** if an error occurred.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_DROP

This macro drops the specified array element from an flist. The element flist is destroyed and the memory reallocated.

> **Note:**
>
> This opcode causes an array to shift its indexing if an element other than the last is dropped. Do not use this PIN_FLIST_ELEM_DROP in a loop of PIN_FLIST_ELEM_GET_NEXT calls; the off-set will cause elements to be skipped.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_ELEM_DROP(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                int32            elem_id,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist containing the array element being removed.

fld
The field number of the array containing the element being removed.

elem_id
The element ID of the element being removed.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_GET

This macro gets the value of a specific array element from the flist. The element remains on the flist unchanged, and the value returned is a pointer to the element flist owned by the flist. The element flist returned *must* be treated as read-only to maintain the integrity of the flist. If a writable copy of the element flist is needed, the application must either make a copy of the returned element flist or use PIN_FLIST_ELEM_TAKE to take ownership of the element from the flist.

Syntax

```
#include "pcm.h"
pin_flist_t *
PIN_FLIST_ELEM_GET(
                pin_flist_t      *flistp,
                pin_fld_num_t     fld,
                int32             elem_id,
                int32             optional,
                pin_errbuf_t     *ebufp);
```

Parameters

flistp
A pointer to the flist containing the array element being obtained.

fld
The field number of the array containing the element being obtained.

elem_id
The ID of the array you need returned.

optional
If this flag is set (by passing in a nonzero value) and the element is not found, no error condition is set. If this flag is not set, and the element is not found, an error condition is set.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the element flist. Returns **NULL** if an error occurred.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_GET_NEXT

This macro gets an array element from an flist. That is, this macro gets the value of the *next* element of a specified array on an flist. It lets the application walk the flist, retrieving each element of an array without knowing the element IDs ahead of time.

The element remains on the flist unchanged, and the value returned is a pointer to the element flist owned by the flist. The element flist returned *must* be treated as read-only to maintain the integrity of the flist. If a writable copy of the element flist is needed, the application must either make a copy of the returned element flist or use PIN_FLIST_ELEM_TAKE_NEXT to take ownership of the element from the flist.

Syntax

```
#include "pcm.h"
pin_flist_t *
PIN_FLIST_ELEM_GET_NEXT(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                int32           *elem_idp,
```

```
                int32           optional,
                pin_cookie_t    *cookie,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist containing the array element being obtained.

fld
The field number of the array containing the element being taken.

elem_idp
A pointer to the number of the array element being taken.

optional
If this flag is set (by passing in a nonzero value) and the element is not found, no error condition is set. If this flag is not set and the element is not found, an error condition is set.

cookie
If set to **NULL**, the first element on the list is returned. In subsequent calls to this macro, pass in the cookie, and the next element of the array is retrieved.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the element flist, *elem_idp*, as the element number. Returns **NULL** if an error occurred or if the element is not found.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_MOVE

This macro moves an element of an array from one flist to another. You can change the field name and record ID when you move the element. The type must remain the same.

Syntax

```
#include "pcm.h"
int32
PIN_FLIST_ELEM_MOVE(
                pin_flist_t     *src_flistp,
                pin_fld_num_t    src_fld,
                pin_rec_id_t     src_recID,
                pin_flist_t     *dest_flistp,
                pin_fld_num_t    dest_fld,
                pin_rec_id_t     dest_recID,
                pin_errbuf_t    *ebufp );
```

Parameters

src_flistp
A pointer to the source flist from which the element is moved.

src_fld
The element that is moved from the source flist.

src_recID
The record ID of the element that is moved.

dest_flistp
A pointer to the destination flist to which an element is moved.

dest_fld
The moved element in the destination flist.

dest_recID
The record ID of the moved element in the destination flist.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns **1** if the field to be moved is found. Returns **0** if the field to be moved is not found. Not finding a field does not result in an error buffer error.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_PUT

This macro puts an array element on an flist. The element flist provided is used as the value of the array element. Ownership of the element flist is passed to the target flist, so the application must not destroy it once it has been put. The memory holding the value must be dynamically allocated.

After the field value has been added to an flist using this macro, the caller can no longer access the value directly using the pointer to the value. The flist management system may optimize memory usage by moving where the value is stored so the original pointer is no longer valid.

If the specified array element already exists on the flist, the existing element flist is destroyed and replaced by the new element flist.

If an error condition exists or this macro fails, the element being put is destroyed. The memory is deallocated, and an error is returned to the error buffer.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_ELEM_PUT(
                pin_flist_t     *flistp,
                pin_flist_t     *elem_flistp,
```

```
                pin_fld_num_t    fld,
                int32            elem_id,
                pin_errbuf_t    *ebufp);
```

Compilation Switch

-DASSIGN_NULL_AFTER_ELEM_PUT
(Release 15.0.1 or later) Assigns *elem_flistp* to a NULL value after the macro call. This prevents the object or flist from being destroyed.

Parameters

flistp
A pointer to the destination flist.

elem_flistp
A pointer to the flist containing the array element being added.

fld
The field number of the array receiving the element.

elem_id
The number of the element to put on the flist.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_SET

This macro sets a copy of an element on an flist. A dynamic copy of the specified element is made for the flist. The element passed in does not have to be in dynamic memory. The element passed in is unaffected by this macro. If the specified element already exists on the flist, the existing element is destroyed and replaced by the new element.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_ELEM_SET(
                pin_flist_t     *flistp,
                void            *elem_flistp,
                pin_fld_num_t    fld,
                int32            elem_id,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the destination flist for the element.

elem_flistp
A pointer to the flist for the input element.

fld
The field number of the array receiving the element.

elem_id
The number of the element being added.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_TAKE

This macro takes the value of an array element from an flist and removes it from the flist. The dynamically allocated memory holding the element flist is returned to the application. The application is then responsible for freeing this element flist when it is no longer needed. This macro is useful when the array element is no longer needed on the flist after the value is retrieved.

Syntax

```
#include "pcm.h"
pin_flist_t *
PIN_FLIST_ELEM_TAKE(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                int32            elem_id,
                int32            optional,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist containing the element being taken.

fld
The field number of the array whose element is being taken.

elem_id
The number of the element being taken.

optional
If this flag is set (by passing in a nonzero value) and the element is not found, no error condition is set. If this flag is not set and the element is not found, an error condition is set.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the element flist. Returns **NULL** if an error occurred or the element is not found.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_ELEM_TAKE_NEXT

This macro takes the value of the *next* element of an array from the flist. It lets the application walk the flist, retrieving each element of an array without knowing the element IDs ahead of time.

The element is removed from the flist. The dynamically allocated memory holding the element flist is returned to the application. The application is then responsible for freeing this element flist when it is no longer needed by the application. This macro is useful when the array element will not be needed on the flist after the value is retrieved.

Syntax

```
#include "pcm.h"
pin_flist_t *
PIN_FLIST_ELEM_TAKE_NEXT(
                pin_flist_t      *flistp,
                pin_fld_num_t     fld,
                int32            *elem_idp,
                int32             optional,
                pin_cookie_t     *cookie,
                pin_errbuf_t     *ebufp);
```

Parameters

flistp
A pointer to the flist of the array containing the element being taken.

fld
The field number of the array containing the element being taken.

elem_idp
A pointer to the number of the element being taken.

optional
If this flag is set (by passing in a nonzero value) and the element is not found, no error condition is set. If this flag is not set and the element is not found, an error condition is set.

cookie
If set to **NULL**, the first element on the list is returned. In subsequent calls to this macro, pass in the cookie, and the next element of the array is retrieved.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the element flist, *elem_idp*, as the element number. Returns **NULL** if an error occurred or if the element is not found.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_FLD_COPY

This macro copies a field from one flist to another. If this macro is called to copy an array, it copies the array with all the elements in the array.

You can change the field name while copying the field. The type must remain the same.

Syntax

```
#include "pcm.h"
int32
PIN_FLIST_FLD_COPY(
                pin_flist_t     *src_flistp,
                pin_fld_num_t    src_fld,
                pin_flist_t     *dest_flistp,
                pin_fld_num_t    dest_fld,
                pin_errbuf_t    *ebufp);
```

Parameters

src_flistp
A pointer to the source flist from which the field is copied.

src_fld
The field that is copied from the source flist.

dest_flistp
A pointer to the destination flist to which a field is copied.

dest_fld
The copied field in the destination flist.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns **1** if the field to be moved is found. Returns **0** if the field to be moved is not found. Not finding a field does not result in an error buffer error.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_FLD_DROP

This macro removes a field from an flist, destroying the value of the field and reallocating the memory.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_FLD_DROP(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist containing the substructure.

fld
The field number of the substructure being removed.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_FLD_GET

This macro gets the value of a field from an flist. The value returned is a pointer to the actual value owned by the flist, and the field remains on the original flist, unchanged. The value returned must be treated as read-only to maintain the integrity of the flist. If a writable copy of the value is needed, the application must either make a copy of the returned value or use PIN_FLIST_FLD_TAKE to take ownership of the field from the flist.

> **Caution:**
>
> The pointer returned is valid only until you modify the flist by setting a field, retrieving a field, or destroying the flist. To ensure that you have a valid pointer, always use PIN_FLIST_FLD_GET immediately before you use the field, or dereference the pointer returned from PIN_FLIST_FLD_GET and store the value locally.

> **Note:**
>
> To copy a field from one flist to another, use PIN_FLIST_FLD_COPY instead of PIN_FLIST_FLD_GET and PIN_FLIST_FLD_SET. To copy an element from one flist to another, use PIN_FLIST_ELEM_COPY.

Syntax

```
#include "pcm.h"
void *
PIN_FLIST_FLD_GET(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                int32            optional,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist containing the field being obtained.

fld
The number of the field being obtained.

optional
If this flag is set (by passing in a nonzero value) and the element is not found, no error condition is set. If this flag is not set and the element is not found, an error condition is set.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the value on the flist. The pointer must be cast appropriately depending on the type of the field. Returns **NULL** if an error occurred or if the field is not found.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_FLIST_FLD_MOVE

This macro moves a field from one flist to another. If this macro is called to move an array, it moves the array with all the elements in the array.

You can change the field name while moving the field. The type must remain the same.

Syntax

```
#include "pcm.h"
int32
PIN_FLIST_FLD_MOVE(
                pin_flist_t     *src_flistp,
                pin_fld_num_t    src_fld,
                pin_flist_t     *dest_flistp,
                pin_fld_num_t    dest_fld,
                pin_errbuf_t    *ebufp );
```

Parameters

src_flistp
A pointer to the source flist from which a field is moved.

src_fld
The field that is moved from the source flist.

dest_flistp
A pointer to the destination flist into which a field is moved.

dest_fld
The moved field in the destination flist.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns **1** if the field to be moved is found. Returns **0** if the field to be moved is not found. Not finding a field does not result in an error buffer error.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_FLD_PUT

This macro puts a field (including its data value) in an flist. The memory holding the value must be dynamically allocated. The dynamic memory holding the value is given to the flist as part of the put. This is useful for adding a field to the flist without copying its value if the application no longer needs that memory.

> **Note:**
>
> To move fields between flists or to rename fields, use PIN_FLIST_FLD_MOVE, PIN_FLIST_ELEM_MOVE, and PIN_FLIST_FLD_RENAME instead of PIN_FLIST_FLD_TAKE and PIN_FLIST_FLD_PUT.

After the field value has been added to an flist using this macro, the caller can no longer access the value directly using the pointer to the value. The flist management system may

optimize memory usage by moving where the value is stored so the original pointer is no longer valid.

If the specified field already exists in the flist, the previous value is destroyed and replaced by the new value.

If an error condition exists or this macro fails, the field being put is destroyed. The memory is deallocated, and an error is returned to the error buffer.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_FLD_PUT(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                void            *valp,
                pin_errbuf_t    *ebufp);
```

Compilation Switch

-DASSIGN_NULL_AFTER_FLD_PUT
(Release 15.0.1 or later) Assigns *valp* to a NULL value after the macro call. This prevents the object or flist from being destroyed.

Parameters

flistp
A pointer to the flist receiving the field.

fld
The number of the field being added.

valp
A pointer to the field value being added.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_FLIST_FLD_RENAME

This macro changes the name of a field in an flist. If you are changing the name of an array, this macro changes the names of all the elements in the array.

The type of the fields must be the same.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_FLD_RENAME(
                pin_flist_t     *flistp,
                pin_fld_num_t    src_fld,
                pin_fld_num_t    dest_fld,
                pin_errbuf_t    *ebufp)
```

Parameters

flistp
A pointer to the flist in which a field is renamed.

src_fld
The field that is renamed.

dest_fld
The new name of the field.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

If the field is not found, the error buffer contains a PIN_ERR_NOT_FOUND error.

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_FLD_SET

This macro adds a field and a value to an flist. A dynamic copy of the specified value is made for the flist. The value passed does not have to be in dynamic memory. The value passed is unaffected by the macro.

If the specified field already exists in the flist, the existing value is destroyed and replaced by the new value.

> **Note:**
>
> To copy a field from one flist to another, use PIN_FLIST_FLD_COPY instead of PIN_FLIST_FLD_GET and PIN_FLIST_FLD_SET. To copy an element from one flist to another, use PIN_FLIST_ELEM_COPY.

Syntax

```
#include "pcm.h"
void
```

```
PIN_FLIST_FLD_SET(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                void            *valp,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist receiving the field.

fld
The number of the field being added.

valp
A pointer to the field value.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_FLIST_FLD_TAKE

This macro takes a field from an flist and returns its value. The dynamically allocated memory holding the field value is returned to the application. The application is then responsible for freeing this memory when it is no longer needed. This macro is useful when fields will not be needed after the field value is retrieved.

> **Caution:**
>
> If you use PIN_FLIST_FLD_GET, you should do so before using this macro. PIN_FLD_FLIST_TAKE can modify the memory locations of the flist, making the PIN_FLIST_FLD_GET pointer invalid. To ensure that the pointer to the flist remains valid, always call PIN_FLIST_FLD_GET immediately before using the field.

Use PIN_FLIST_FLD_GET when a read-only pointer to the field is needed.

> **Note:**
>
> To move fields between flists or to rename fields, use PIN_FLIST_FLD_MOVE, PIN_FLIST_ELEM_MOVE, and PIN_FLIST_FLD_RENAME instead of PIN_FLIST_FLD_TAKE and PIN_FLIST_FLD_PUT.

Syntax

```
#include "pcm.h"
void *
PIN_FLIST_FLD_TAKE(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                int32            optional,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist containing the field being taken.

fld
The number of the field being taken.

optional
If this flag is set (by passing in a nonzero value) and the element is not found, no error condition is set. If this flag is not set and the element is not found, an error condition is set.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the field's value. The pointer must be cast appropriately depending on the type of field. Returns **NULL** if an error occurred or if the field is not found.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_SUBSTR_ADD

This macro adds a substructure to an flist. The flist for the substructure is created and returned. The pointer to this substruct flist can then be used to set/put fields into the substructure. If the substructure already exists on the flist, the existing substruct flist is destroyed and replaced by the new substruct flist.

Syntax

```
#include "pcm.h"
pin_flist_t *
PIN_FLIST_SUBSTR_ADD(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist receiving the substructure.

fld
The field number of the substructure being added.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the flist for the substructure. Returns **NULL** if an error occurred.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_SUBSTR_DROP

This macro removes a substructure from an flist, freeing the allocated memory.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_SUBSTR_DROP(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist containing the substructure being dropped.

fld
The field number of the substructure being dropped.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_SUBSTR_GET

This macro gets a substructure from an flist. The substructure remains on the flist unchanged, and the value returned is a pointer to the substructure flist, owned by the flist. The substructure

returned *must* be treated as read-only to maintain the integrity of the flist. If a writable copy of the substructure flist is needed, the application must either make a copy of the returned substructure flist or use the PIN_FLIST_SUBSTR_TAKE macro to take ownership of the substructure.

Syntax

```
#include "pcm.h"
void *
PIN_FLIST_SUBSTR_GET(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                int32            optional,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist with the substructure being obtained.

fld
The field number of the substructure being obtained.

optional
If this flag is set (by passing in a nonzero value) and the element is not found, no error condition is set. If this flag is not set and the element is not found, an error condition is set.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the substructure flist. Returns **NULL** if an error occurred or if the element is not found.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_SUBSTR_PUT

This macro puts a substructure on an flist. The substructure flist provided is used as the value of the substructure. Ownership of the substructure flist is passed to the target flist, so the application must not destroy it once it has been put. The memory holding the value must be dynamically allocated.

After the value of the field has been added to an flist using this macro, the caller can no longer access the value directly using the pointer to the value. The flist management system may optimize memory usage by moving where the value is stored, so the original pointer is no longer valid.

If the specified substructure already exists on the target flist, the existing element is destroyed and replaced by the new element.

If an error condition exists or the macro otherwise fails, the substructure being put is destroyed. The memory is deallocated and an error is returned to the error buffer.

This macro is optimal for adding inordinately large chunks of data to an flist. The flist does not allocate memory for the added data; it is merely linked to where the memory is already dynamically allocated. In contrast, PIN_FLIST_SUBSTR_SET adds an element by reallocating memory for it in the flist.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_SUBSTR_PUT(
                pin_flist_t     *flistp,
                void            *substr_flistp,
                pin_fld_num_t    fld,
                pin_errbuf_t    *ebufp);
```

Compilation Switch

-DASSIGN_NULL_AFTER_SUBSTR_PUT
(Release 15.0.1 or later) Assigns *substr_flistp* to a NULL value after the macro call. This prevents the object or flist from being destroyed.

Parameters

flistp
A pointer to the flist being added.

substr_flistp
A pointer to the flist containing the substructure being added.

fld
The field number of the substructure being added.

ebufp
A pointer to the error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_SUBSTR_SET

This macro adds a copy of a substructure to an flist. A dynamic copy of the specified substructure is made for the flist. The substructure passed in does not have to be in dynamic memory. The substructure passed in is unaffected by this macro. If the specified field already exists on the flist, the existing substructure is destroyed and replaced by the new substructure.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_SUBSTR_SET(
                pin_flist_t     *flistp,
                void            *substr_flistp,
```

```
                pin_fld_num_t    fld,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist receiving the substructure.

substr_flistp
A pointer to the flist containing the substructure being added.

fld
The field number of the substructure being added.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_SUBSTR_TAKE

This macro takes a substructure off of an flist and returns its value. The dynamically allocated memory holding the field value is returned to the application. The application is then responsible for freeing this memory when it is no longer needed. This macro is useful when fields will not be needed after the field value is retrieved.

Syntax

```
#include "pcm.h"
void *
PIN_FLIST_SUBSTR_TAKE(
                pin_flist_t     *flistp,
                pin_fld_num_t    fld,
                int32            optional,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist containing the substructure being taken.

fld
The field number of the substructure being removed from *flistp*.

optional
If this flag is set (by passing in a nonzero value) and the element is not found, no error condition is set. If this flag is not set and the element is not found, an error condition is set.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Flist Management Macros

This section describes flist management macros.

PIN_FLIST_CONCAT

This BRM macro appends a (source) flist to the end of another (destination) flist. No comparisons between the flists are performed, and the source flist remains unchanged.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_CONCAT(
                pin_flist_t     *dest_flistp,
                pin_flist_t     *src_flistp,
                pin_errbuf_t    *ebufp);
```

Parameters

dest_flistp
A pointer to the destination flist.

src_flistp
A pointer to the source flist.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns the concatenated flist in *dest_flistp*. If *src_flistp* is **NULL**, *dest_flistp* is returned unchanged. Returns an error in the error buffer if *dest_flistp* is **NULL**.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_COPY

This macro copies all levels of an existing flist, including its array elements and substructures. The copied fields and their values are duplicated so no memory is shared between the two flists.

Syntax

```
#include "pcm.h"
pin_flist_t *
PIN_FLIST_COPY(
                pin_flist_t     *flistp,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist to be copied.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the new flist. Returns **NULL** if an error occurred.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_COUNT

This macro counts the number of fields on the flist. Only fields on the main flist are included. Each array element and substruct is counted as a single element.

If PIN_FLIST_COUNT is called with the pointer to an array element or substruct, the number of fields at that level of the flist are counted.

Syntax

```
#include "pcm.h"
int32
PIN_FLIST_COUNT(
                pin_flist_t     *flistp,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to an flist to count the fields of.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns the number of fields as an unsigned integer. Returns **0** if an error occurred.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_CREATE

This BRM macro creates an flist that is used to pass parameters to the PCM_OP function. This macro creates an flist and returns a pointer that is used to reference the flist by all future operations. All memory for the flist is dynamically allocated.

Syntax

```
#include "pcm.h"
pin_flist_t *
PIN_FLIST_CREATE(ebufp)
                pin_errbuf_t     *ebufp);
```

Parameter

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the flist, in the form of **pin_flist_t***. Returns **NULL** if an error occurred.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_FLIST_DESTROY

This macro destroys an flist. Flists use dynamically allocated memory, and they must be destroyed to free that memory. This macro destroys the entire contents of an flist, including all fields on the flist.

PIN_FLIST_DESTROY can destroy an flist, even if the error buffer is **NULL**.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_DESTROY(
```

```
                pin_flist_t     *flistp,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist to destroy.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_FLIST_DESTROY_EX

This macro destroys an flist. Flists use dynamically allocated memory, and they must be destroyed to free that memory. This macro first checks whether the pointer passed in is **NULL**. If the pointer is **NULL**, it returns. If the pointer is not **NULL**, it destroys the entire contents of the flist, including all fields on the flist, and sets the flist pointer to **NULL**.

> **Note:**
>
> PIN_FLIST_DESTROY_EX can destroy an flist, even if the error buffer is **NULL**.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_DESTROY_EX(
                pin_flist_t     **flistpp,
                pin_errbuf_t     *ebufp);
```

Parameters

****flistpp***
A pointer to the flist to destroy.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_FLIST_PRINT

This macro prints, in ASCII format, an flist to a file. All levels of the flist, including the contents of array elements and substructures, are printed. This is useful for debugging applications that build or manipulate flists.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_PRINT(
                pin_flist_t     *flistp,
                FILE            *fi,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist to print.

fi
A pointer to a file to print a message to. If the value of this pointer is **NULL**, the message is printed to stdout.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Example

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_FLIST_SORT

This macro sorts flists and is normally used to sort array elements. Arrays sorted may also be the result of a search.

The flist to be sorted usually represents an array of search results returned from PCM_OP_SEARCH. The **sort_flistp** parameter is an flist that you construct with **sort_parameter**, called PIN_FLD_RESULTS. It would look like:

```
PIN_FLD_RESULTS
        field 1
        field 2
          .
          .
          .
```

Then use *sort_default* to compare nonexistent fields to existing fields. If all of the result elements have field values, **0** can be passed as the value of *sort_default*.

In cases where a result element has a field value, and it is being compared to another result element with the same field, but no value:

- A negative *sort_default* means that the result element with the missing field value is sorted *before* the other in the sorted list.
- A positive *sort_default* means the missing field occurs *after* the other.
- A *sort_default* of **0** means that they are considered equal and order is arbitrary on the sorted list.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_SORT(
                pin_flist_t     *flistp,
                pin_flist_t     *sort_listp,
                int32            sort_default,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist being sorted. The flist should normally consist of an array so that the sort is performed on elements of the array. Each element of the array may be a list of fields; it is those fields that get sorted. When you call this macro, pass the exact array (flist) you want sorted, not the entire array.

sort_listp
A list of fields in each element in *flistp* to use as sort fields. Elements in *flistp* are sorted in this order. If the value of this parameter is **NULL**, PIN_ERR_BAD_ARG is returned.

sort_default
The comparison to be used if an element is not found:

- f1 NOT found, f2 found - return *sort_default*
- f1 found, f2 NOT found - return *-sort_default*
- f1 NOT found, f2 NOT found - return 0 (equal)
- a negative value for *sort_default* means: f1 < f2
- a positive value for *sort_default* means: f1 > f2
- a zero value for *sort_default* means: f1 == f2

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_SORT_REVERSE

This macro sorts flists in reverse order. This macro, along with PIN_FLIST_SORT, is normally used to sort array elements. Arrays sorted may also be the result of a search.

The flist to be sorted usually represents an array of search results returned from PCM_OP_SEARCH or PCM_OP_STEP_SEARCH. The **sort_flistp** parameter is an flist that you construct with **sort_parameter**, called PIN_FLD_RESULTS. It would look like:

```
PIN_FLD_RESULTS
        field n
          .
          .
          .
        field 2
        field 1
```

Then use the *sort_default* parameter to compare nonexistent fields to existing fields. If all of the result elements have field values, **0** can be passed as the value of *sort_default*.

In cases where a result element has a field value, and it is being compared to another result element with the same field, but no value:

- A negative *sort_default* means that the result element with the missing field value is sorted *after* the other in the sorted list.
- A positive *sort_default* means the missing field occurs *before* the other.
- A *sort_default* of **0** means that they are considered equal and order is arbitrary on the sorted list.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_SORT_REVERSE(
                pin_flist_t     *flistp,
                pin_flist_t     *sort_listp,
                int32            sort_default,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist being sorted. The flist should normally consist of an array so that the sort is performed on elements of the array. Each element of the array may be a list of fields; it is those fields that get sorted.

sort_listp
A list of fields in each element in *flistp* to use as sort fields. Elements in *flistp* are sorted in this order. If the value of this parameter is **NULL**, PIN_ERR_BAD_ARG is returned.

sort_default
The comparison to be used if an element is not found:

- a zero value for *sort_default* means: f1 == f2
- a positive value for *sort_default* means: f1 > f2
- a negative value for *sort_default* means: f1 < f2
- f1 NOT found, f2 NOT found - > return 0 (equal)
- f1 found, f2 NOT found -> return -*sort_default*
- f1 NOT found, f2 found -> return *sort_default*

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_STR_TO_FLIST

This macro takes a string representation of an flist (for example, the output of PIN_FLIST_TO_STR) and creates an flist run-time data structure.

Syntax

```
#include "pcm.h"
void
```

```
PIN_STR_TO_FLIST(
                char            *str,
                int64            default_db,
                pin_flist_t     **flistp,
                pin_errbuf_t     *ebufp);
```

Parameters

str
A pointer to a string containing an flist in ASCII form.

default_db
A specified database number. If the ASCII string contains the sub-string "$DB", the database number in this parameter will replace it.

flistp
A pointer to a buffer for the return flist.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns the string in *flistp*.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_TO_STR

This macro prints, in ASCII format, the contents of an flist to a buffer.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_TO_STR(
                pin_flist_t     *flistp,
                char            **strpp,
                int32            *lenp,
                pin_errbuf_t     *ebufp);
```

Parameters

flistp
A pointer to the flist to print to a string.

strpp
A pointer to a buffer for the return string. If the value is **NULL**, a buffer is allocated using malloc.

lenp
The length of the buffer that *strpp* points to. The buffer must be large enough to include a **\0**. If the value of *strpp* is **NULL**, *len* is passed back as the size of the allocated buffer, including the **\0**.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns the string in *strpp*. If a buffer was allocated, *len* is the size of the string, including the **NULL** terminator. If a buffer is allocated, the application owns the memory and must free it eventually.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_TO_STR_COMPACT_BINARY

This macro prints, in compact binary form, the contents of an flist to a buffer.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_TO_STR_COMPACT_BINARY(
                pin_flist_t     *flistp,
                char           **strpp,
                int32           *lenp,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist to print to a string.

strpp
A pointer to a buffer for the return string. If the value is **NULL**, a buffer is allocated using malloc.

lenp
The length of the buffer that *strpp* points to. The buffer must be large enough to include a **\0**. If the value of *strpp* is **NULL**, *len* is passed back as the size of the allocated buffer, including the **\0**.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns the string in *strpp*. The string is stored in binary format in compact form, which means the field numbers, instead of the field names, are stored in the buffer. If a buffer was allocated,

len is the size of the string, including the **NULL** terminator. If a buffer is allocated, the application owns the memory and must free it eventually.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_FLIST_TO_XML

This macro converts an flist to XML format. It is designed for converting an invoice to an XML format. The formatted XML invoice is generated directly from the flist. It ignores and does not convert data in buffer fields or fields of type PIN_FLDT_BINSTR.

> **Note:**
>
> This macro does not generate a **.DTD** file.

Syntax

```
#include "pcm.h"
void
PIN_FLIST_TO_XML(
                pin_flist_t     *flistp,
                int32            flags,
                int32            encoding,
                char           **bufpp,
                int             *lenp,
                char            *root_elemname,
                pin_errbuf_t    *ebufp);
```

Parameters

flistp
A pointer to the flist to convert.

flags
Specifies the name-attribute pairs to use for the XML element tag:

- PIN_XML_BY_TYPE
- Uses the **TYPE** field for the name of the XML element tag. This is the default.
- PIN_XML_BY_NAME
- Uses the field name for the name of the XML element tag.
- PIN_XML_BY_SHORT_NAME
- Uses the field name for the name of the XML element tag and drops the common prefix to include only the unique portion. For example, PIN_FLD_NAME becomes NAME.
- PIN_XML_FLDNO
- Uses the field number for the attribute of the XML element tag.

- PIN_XML_TYPE

  Uses the **TYPE** field for the attribute of the XML element tag.

encoding
Specify **UTF8**.

bufpp
A pointer to the buffer that will contain the XML converted data.

lenp
The size of the buffer that *bufpp* points to.

root_elemname
The root element name. If you do not specify this field, the default root element name, **document**, is used.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

POID Management Macros

This section describes POID management macros.

PIN_POID_COMPARE

This BRM macro compares two POIDs for equality. All fields of the POIDs, including the revision level, must be identical for them to be considered equal.

Syntax

```
#include "pcm.h"
int32
PIN_POID_COMPARE(
                poid_t          *poidp1,
                poid_t          *poidp2,
                int32            check_rev,
                pin_errbuf_t    *ebufp);
```

Parameters

poidp1
A pointer to the first POID to be compared.

poidp2
A pointer to the second POID to be compared.

check_rev
Determines whether or not the revision level of two POIDs is compared. If *check_rev* is set to **0**, only the POID ID, database number, and type are compared. If *check_rev* is set to a nonzero value, the POID ID, database number, type, and revision number are compared.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns **0** if the POIDs are identical. Returns a negative value if *poidp1* is less than *poidp2*. Returns a positive value if *poidp1* is greater than *poidp2*.

Error Handling

This routine uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer, and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_COPY

This macro copies a POID. The new POID uses dynamically allocated memory and is owned by the caller.

If *src_poidp* is **NULL**, or if the source POID data **type** is **NULL**, a **NULL** value is returned, and no error condition is set.

Syntax

```
#include "pcm.h"
poid_t*
PIN_POID_COPY(
               poid_t           *src_poidp,
               pin_errbuf_t     *ebufp);
```

Parameters

src_poidp
A pointer to the source POID.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the newly created POID if the macro is successful. Returns **NULL** if the macro fails.

Success codes

PCM_ERR_NONE

Error codes

PCM_ERR_NO_MEM

Error Handling

This routine uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer, and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_CREATE

This macro creates a POID. The POID uses dynamically allocated memory, and ownership of the POID is given to the caller. A copy is made of *type*, so it does not need to be in dynamic memory when passed.

id is typically initialized as **0**. The create operation finds the next available ID in the database and uses it when creating the object.

A source POID with a *type* of **NULL** is handled correctly. See "Portal Object ID (POID)" in *BRM Developer's Guide* for more information.

Syntax

```
#include "pcm.h"
poid_t*
PIN_POID_CREATE(
                int64             db,
                char             *type,
                int64             id,
                pin_errbuf_t     *ebufp);
```

Parameters

db
The database number.

type
The data type for the new POID. See the list of objects in "Storable Class Definitions". Examples are **/service** and **/event/customer/nameinfo**.

id
A unique object ID. This is a 64-bit quantity, so an extremely large number of objects can exist within a single database. Object IDs are unique within a single database, but not across databases.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the newly created POID if the macro is successful. Returns **NULL** if the macro fails.

Error Handling

This routine uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer, and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Examples

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_POID_DESTROY

This macro destroys a POID. POIDs use dynamically allocated memory and must be destroyed to free that memory. The entire POID is destroyed, including the **type** string.

Syntax

```
#include "pcm.h"
void
PIN_POID_DESTROY(
                poid_t          *poidp,
                pin_errbuf_t    *ebufp);
```

Parameters

poidp
A pointer to the POID to be destroyed.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller. This parameter is optional. If a **NULL** is passed in, no error information is returned.

Return Values

This macro returns nothing.

Error Handling

This routine uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer, and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

Examples

The **sample_app.c** file and the accompanying makefile illustrate how to use this macro when setting up a generic BRM account and service. The files are located in *BRM_SDK_home***/source/samples/app/c**.

PIN_POID_FROM_STR

This macro converts a string to a POID.

> **Note:**
>
> This macro allocates the new POID's memory. To avoid memory leaks, PUT the POID onto an flist (typical case) or destroy the flist.

Syntax

```
#include "pcm.h"
poid_t*
PIN_POID_FROM_STR(
               char             *strp,
               char            **endcpp,
               pin_errbuf_t     *ebufp);
```

Parameters

strp
A pointer to the destination string.

endcpp
A pointer to the character following the last character of the POID value. That is, the character that terminated the scan (usually **NULL**, white space, or a new line).

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

Returns a pointer to the POID created from the input string if the macro is successful. Returns **NULL** if the macro fails.

Error Handling

This routine uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer, and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_GET_DB

This macro returns the database number portion of a POID.

Syntax

```
#include "pcm.h"
int64
PIN_POID_GET_DB(
               poid_t     *poidp);
```

Parameter

poidp
A pointer to the POID whose database number is being returned.

Return Values

Returns the database number if the macro is successful.

Error Handling

This macro does not handle errors.

PIN_POID_GET_ID

This macro returns a POID's ID.

Syntax

```
#include "pcm.h"
int64
PIN_POID_GET_ID(
               poid_t     *poidp);
```

Parameter

poidp
A pointer to the POID whose ID is being returned.

Return Values

Returns the POID's ID if the macro is successful.

Error Handling

This macro does not handle errors.

PIN_POID_GET_REV

This macro returns the POID's revision level. The revision level is incremented each time any portion of the object is updated.

Syntax

```
#include "pcm.h"
int32
PIN_POID_GET_REV(
               poid_t     *poidp);
```

Parameter

poidp
A pointer to the POID whose nonzero revision level is being returned.

Return Values

Returns the POID's revision level if the macro is successful.

Error Handling

This macro does not handle errors.

PIN_POID_GET_TYPE

This macro returns the object type of the POID in string format. Possible types are listed in "Storable Class Definitions". Examples are **/account** and **/event/billing/charge**.

Syntax

```
#include "pcm.h"
char*
PIN_POID_GET_TYPE(
               poid_t     *poidp);
```

Parameter

poidp
A pointer to the POID whose type is being returned.

Return Values

Returns the POID's type as a string if the macro is successful.

Error Handling

This macro does not handle errors.

PIN_POID_IS_NULL

This macro checks a POID to see whether it is **NULL**. The condition is satisfied if the pointer is **NULL** or the database number is **0**.

Syntax

```
#include "pcm.h"
int32
PIN_POID_IS_NULL(
               poid_t     *poidp);
```

Parameter

poidp
A pointer to the POID to check.

Return Values

Returns a nonzero value if the POID pointer is **NULL** or the database number is **0**.

Error Handling

This macro does not handle errors.

PIN_POID_LIST_ADD_POID

This macro adds a POID to the POID list.

Syntax

```
#include "pcm.h"
void
PIN_POID_LIST_ADD_POID(
                      char           **strpp,
                      poid_t         *pdp,
                      int32          flag,
                       pin_errbuf_t   *ebufp)
```

Parameters

strpp
Pointer to the POID list.

pdp
Pointer to the POID to be added to the list.

flag
A PCM flag (PCM_FLDFLG_FIFO or PCM_FLDFLG_CMPREV).

ebufp
Pointer to the error buffer.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_LIST_COPY

This macro copies a POID list.

Syntax

```
#include "pcm.h"
poid_list_t *
PIN_POID_LIST_COPY(
                   poid_list_t  *src_pldp,
                    pin_errbuf_t  *ebufp)
```

Parameters

src_pldp
Pointer to the POID list to be copied.

ebuf
Pointer to the error buffer.

Return Values

Returns a pointer to the newly created POID list if the macro is successful. Returns NULL if the macro fails.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_LIST_COPY_NEXT_POID

This macro copies 'next' POID from the POID list.

Syntax

```
#include "pcm.h"
poid_t *
pin_poid_list_get_next(
                      char            *strp,
                      int32           optional,
                      pin_cookie_t    *cookiep,
                       pin_errbuf_t    *ebufp)
```

Parameters

strp
Pointer to the POID list from which the next POID is to be copied.

optional
If this flag is set to a nonzero value and the element is not found, no error condition is set. If this flag is not set, and the element is not found, an error condition is set.

cookiep
The cookie for the next POID.

ebufp
Pointer to the error buffer.

Return Values

Returns a pointer to the newly created POID if the macro is successful. Returns NULL if the macro fails.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_LIST_COPY_POID

This macro copies the specified POID from the POID list.

Syntax

```
#include "pcm.h"
poid_t*
PIN_POID_LIST_COPY_POID(
                       char            *strp,
                       void            *vp,
                       int32           flags,
                        pin_errbuf_t    *ebufp)
```

Parameters

strpp
Pointer to the POID list.

vp
Pointer to the POID to be copied.

flags
A PCM flag (PCM_FLDFLG_CMPREV or PCM_FLDFLG_TYPE_ONLY) to check for the existence of the POID to be copied.

Ebufp
Pointer to the error buffer.

Return Values

Returns a pointer to the newly created POID if the macro is successful. Returns NULL if the macro fails.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_LIST_CREATE

This macro creates a POID list.

Syntax

```
#include "pcm.h"
poid_list_t *
PIN_POID_LIST_CREATE(
                     pin_errbuf_t *ebufp)
```

Parameter

ebufp
Pointer to the error buffer.

Return Values

Returns a pointer to the newly created POID list if macro is successful. Returns NULL if the macro fails.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_LIST_DESTROY

This macro frees a POID list.

Syntax

```
#include "pcm.h"
void
PIN_POID_LIST_DESTROY(
                      poid_list_t *pldp,
                        pin_errbuf_t *ebufp)
```

Parameters

pldp
Pointer to the POID list to be freed.

ebufp
Pointer to the error buffer.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_LIST_REMOVE_POID

This macro removes a POID from the POID list.

Syntax

```
#include "pcm.h"
void
PIN_POID_LIST_REMOVE_POID(
                          char        **strpp,
                          poid_t      *pdp,
                          int32       check_rev,
                            pin_errbuf_t *ebufp)
```

Parameters

strpp
Pointer to the POID list.

pdp
Pointer to the POID to be removed from the list.

check_rev
Determines the existence of the POID to be removed. If *check_rev* is set to 0, existence of the POID is checked.

ebufp
Pointer to the error buffer.

Return Values

This macro returns nothing.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_LIST_TAKE_NEXT_POID

This macro takes the 'next' POID from the POID list.

Syntax

```
#include "pcm.h"
poid_t *
pin_poid_list_take_next(
                        char            **strpp,
                        int32           optional,
                           pin_errbuf_t    *ebufp)
```

Parameters

strpp
Pointer to the POID list.

optional
If this flag is set to a nonzero value and the element is not found, no error condition is set. If this flag is not set, and the element is not found, an error condition is set.

ebufp
Pointer to the error buffer.

Return Values

Returns a pointer to the POID taken from the POID list if the macro is successful. Returns NULL if the macro fails.

Error Handling

This macro uses series-style ebuf error handling. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_PRINT

This macro prints a POID.

Syntax

```
#include "pcm.h"
void
PIN_POID_PRINT(
               poid_t           *poidp,
               FILE             *fi,
               pin_errbuf_t     *ebufp);
```

Parameters

poidp
A pointer to the POID to print.

fi
The **FILE** pointer to the file to receive the message. If the value of **FILE** is **NULL**, the message is printed to **stdout**.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This routine uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer, and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

PIN_POID_TO_STR

This macro prints a POID to a string. Put the info of a POID into a string (*strpp*). If the buffer (*ebufp*) is not large enough to hold the string, **PIN_ERR_BAD_ARG** is returned. The return value of *lenp* includes the **\0**. The format of the string is:

"%d %s %d %d"

where the values are for:

database_number object_type object_id object_revision_level

object_revision_level is incremented each time the object is updated.

Syntax

```
#include "pcm.h"
void
PIN_POID_TO_STR(
                poid_t           *poidp,
                char            **strpp,
                int32            *lenp,
                pin_errbuf_t     *ebufp);
```

Parameters

poidp
A pointer to the POID to be printed.

strpp
A pointer to the buffer receiving the string version of the POID. This should be 48 larger than the value of PCM_MAX_POID_TYPE, to accommodate the largest strings.

lenp
The length of the buffer.

ebufp
A pointer to an error buffer. Used to pass status information back to the caller.

Return Values

This macro returns nothing.

Error Handling

This routine uses series-style ebuf error handling. Applications can call any number of series ebuf–style API routines using the same error buffer, and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors. See "Series-Style ebuf" in *BRM Developer's Guide* for more information.

String Manipulation Functions

This section describes string manipulation functions.

About the String Manipulation Functions

You use the string manipulation functions to store and retrieve server strings, such as reason codes, help messages, and other text displayed in the user interface. These strings are stored on the server so that they can be easily localized for multiple languages and displayed simultaneously in the appropriate languages for the client locales. For example, French and German customer service representatives (CSRs) logged into BRM at the same time can read messages in their own languages.

String manipulation functions also allow data received by the database to be canonicalized for easy processing.

BRM Locale IDs

Linux, Windows, and Java use different locale IDs. So BRM includes a locale table, which maps the BRM locale to locale strings for various platforms.

Similar to Linux, the BRM locale is either:

- The two-character ISO code for the language. These two-character locales are used for a language in its country of origin. For example, **fr** designates French used in France.
- A concatenation of the two-character ISO code for the language and the two-character ISO code for the country. For example, **en_US** designates English in the United States.

The locale description IDs are mapped to a **/strings** table containing the textual description of the supported locales. This table and the BRM table name are stored in the database under **/config/locales**.

For more information on BRM locale names, see "Locale Names" in *BRM Developer's Guide*.

Storable Class Hierarchy for Localized Strings

BRM includes a **/strings** storable class to store localized strings.

> **Note:**
>
> You cannot extend the **/strings** storable class.

Structure of the **/strings** storable class:

```
/strings
POID PIN_FLD_POID
TIMESTAMP PIN_FLD_CREATED_T
TIMESTAMP PIN_FLD_MOD_T
STRING PIN_FLD_DOMAIN             required, length = 1023
STRING PIN_FLD_DESCR              optional, length = 1023
STRING PIN_FLD_LOCALE             required, length = 1023
INT PIN_FLD_STRING_ID             required
INT PIN_FLD_STR_VERSION           required
STRING PIN_FLD_STRING             required, length = 1023
STRING PIN_FLD_HELP_STRING        optional, length = 1023
```

For descriptions of the fields, see the **/strings** storable class description.

> **Note:**
>
> Do not change these names and numbers or the information will not be accessible.

Locale Mapping

For detailed information on BRM locale mapping, see "Locale Names" in *BRM Developer's Guide*.

Localized String Data Files

A file of localized string data contains multibyte character set (MBCS) strings, and the data is loaded into the database by running a utility that constructs storable string objects using information in the file.

The file extension of the file must be the BRM locale ID.

Sample names for files containing localized string data:

- **locale_descr.en_US** contains locale description information for United States English.
- **reasons.en_US** contains all of the reason code data for United States English.

String File Format Description

This section describes the required format of the string file. To use this file with the related functions and utilities, the file must follow this format.

> **Note:**
>
> The load utility parser is case-insensitive to the keywords. It passes the locale and domain strings to the database as received. BRM is case sensitive. For example, **en_us** and the BRM locale **en_US** are not considered the same, nor are "Reason Codes-Credit Reasons" and "reason codes-credit reasons."

- Comments begin with the # symbol. All comments and white space are ignored.

- The string file has a locale ID as the first uncommented statement of the file, and there is only one locale ID per file. You can use existing domains in the files and/or add your own. Organize your strings by domains within the file.
- The string object definition is bounded by STR-END and consists of an ID unique within a domain, a string version, and the string itself.
- A string is delimited by quotation marks and can contain any character, including a quotation mark if escaped (\"). The percent symbol followed by an integer (%1) is interpreted as a substitution parameter flag.
- For reason codes, the version field specifies the domain of the reason, such as credit or debit.

This example shows a compatible string file:

```
######################
# strings.en_US
######################

LOCALE = "en_US" ;

DOMAIN = "Reason Codes-Credit Reasons" ;
STR
     ID = 1 ;
     VERSION = 1 ;
     STRING = "Customer not satisfied with service" ;
END
STR
     ID = 2 ;
     VERSION = 1 ;
     STRING = "Customer unaware of charges" ;
END
STR
     ID = 3 ;
     VERSION = 1 ;
     STRING = "Debited account by mistake" ;
END

DOMAIN = "Reason Codes-Debit Reasons" ;
STR
     ID = 1 ;
     VERSION = 1 ;
     STRING = "Technical and support charges" ;
END
STR
     ID = 2 ;
     VERSION = 1 ;
     STRING = "Service charges" ;
END
STR
     ID = 3 ;
     VERSION = 1 ;
     STRING = "Credited account by mistake" ;
END
```

String Manipulation Example

You can create message strings in multiple languages to obtain all the reason codes for English.

This is an example definition:

```
string_list_t*
pcm_get_localized_string_list(
               pcm_context_t     *context_p,
               const char        *locale_p,
               const char        *domain_p,
               const int32        string_id,
               const int32        string_vers,
               pin_errbuf_t      *ebufp);
```

The top-level function, **pcm_get_localized_string_list**, allows arbitrary queries on the **/strings** table. The argument list is similar to **pcm_get_localized_string** except that message buffers are not supplied by the caller. The function can accept a null locale string, a null domain string, a string ID = -1, or a string version = -1 to indicate that the argument is not part of the search.

This example shows retrieving strings:

```
pcm_get_localized_string_list(context_p,"en_US","Reason Codes-Active Status
Reasons",-1,1,ebufp);
```

is equivalent to:

```
select*
from strings_t
where locale = "en_US" AND
     domain = "Reason Codes-Active Status Reasons" AND
     string_vers = 1
```

which returns a set of string objects for any locale ID fitting these criteria. The function returns a container object of type **string_list_t**.

String Manipulation Functions

Table 1-5 lists String Manipulation Functions.

Table 1-5 String Manipulation Functions

Function	Description
pcm_get_localized_string_list	Retrieves the specified string list to be used by the string manipulation functions.
pin_string_list_destroy	Deallocates the object and its flist when finished with the string list.
pin_string_list_get_next	Retrieves the next object in the string list.

pcm_get_localized_string_list

This function retrieves the specified string list to be used by the string manipulation functions.

Use this function to obtain a group of related strings. It is much more efficient than calling **pcm_get_error_message** for each individual string.

pin_string_list_destroy

This function deallocates the object and its flist when finished with the string list.

> **Note:**
>
> To prevent memory leaks, you must call this after calling **pcm_get_string_list**.

Syntax

```
void
pin_string_list_destroy(
                string_list_t     *string_listp,
                pin_errbuf_t      *ebufp);
```

Parameters

string_listp
A pointer to the list.

ebufp
A pointer to an error buffer. Passes status information back to the caller.

pin_string_list_get_next

This function retrieves the next object in the string list.

The caller passes in the string list and a string info object, and the attributes of the next string object are pulled from the list and copied to the string info object. The info object is then returned to the caller. This function calls **pin_string_info_init** internally to flush the string info object and prepare it for new data. This allows the same string info object to be used repeatedly when iterating through the list.

Syntax

```
string_info_t*
pin_string_list_get_next(
                string_list_t     *string_listp,
                string_info_t     *string_infop,
                pin_errbuf_t      *ebufp);
```

Parameters

string_listp
A pointer to the list.

string_infop
A pointer to the string.

ebufp
A pointer to an error buffer. Passes status information back to the caller.

Validity Period Manipulation Macros

Validity period manipulation macros are used to get and set relative offset values for validity periods that start and end after a relative period passes. For example, a charge offer's cycle fee period can become effective three months after the charge offer is purchased.

About Relative Offset Values

Relative validity period information is stored in the BRM database in DETAILS fields. There are DETAILS fields for charge offer, discount, and balance validity periods. The specific name of the fields vary, but all end with "_DETAILS".

Relative validity period information includes the following values:

- Mode - Specifies generally when the validity period starts or ends and can be one of these:
  - PIN_VALIDITY_ABSOLUTE = 0
  - PIN_VALIDITY_IMMEDIATE = 1
  - PIN_VALIDITY_NEVER = 2
  - PIN_VALIDITY_FIRST_USAGE = 3
  - PIN_VALIDITY_RELATIVE = 4
- Unit - Specifies the type of offset unit, which can be one of these:
  - Seconds = 1
  - Minutes = 2
  - Hours = 3
  - Days = 4
  - Months = 5
  - Event cycles = 7
  - Accounting cycles = 8
  - Billing cycles = 9
  - None = 0
- Offset - Specifies the number of units in the offset period.

> **Note:**
>
> Not all of the unit and mode values listed above can be used with every relative validity period in BRM. The unit and mode you can specify depends on the validity period you are setting and whether you are setting the start or end time. For more information, see the following topics:
>
> - For information about the relative start and end times of charge offers and discount offers in bundles, see "About the Validity Periods of Offers in Bundles" in *BRM PDC Creating Product Offerings*.
> - For information about the relative start and end times of discount offers owned by accounts, see "Setting Discount Offer Purchase, Cycle, and Usage Start and End Times" in *BRM Managing Customers*.
> - For information about the relative start and end times of balances, see "Configuring Validity Periods for Noncurrency Credit Balances" in *BRM PDC Creating Product Offerings*.

PIN_VALIDITY_GET_UNIT

This macro retrieves the relative offset unit from the start- or end-time details value that is passed in.

Syntax

```
#include "pcm.h"
u_int32
PIN_VALIDITY_GET_UNIT(
                u_int32     encoded_value);
```

Parameter

encoded_value
The encoded value of the start- or end-time details field.

Return Values

Returns the value of the relative offset unit.

PIN_VALIDITY_GET_OFFSET

This macro retrieves the relative offset (the number of units in the relative period) from the start- or end-time details value that is passed in.

Syntax

```
#include "pcm.h"
u_int32
PIN_VALIDITY_GET_OFFSET(
                u_int32     encoded_value);
```

Parameter

encoded_value
The encoded value of the start- or end-time details field.

Return Values

Returns the value of the relative offset.

PIN_VALIDITY_GET_MODE

This macro retrieves the mode value from the start- or end-time details value that is passed in.

Syntax

```
#include "pcm.h"
pin_validity_modes_t
PIN_VALIDITY_GET_MODE(
                u_int32     encoded_value);
```

Parameter

encoded_value
The encoded value of the start- or end-time details field.

Return Values

Returns the value of the relative mode.

PIN_VALIDITY_SET_UNIT

This macro sets the relative offset unit in the start- or end-time details value that is passed in.

Syntax

```
#include "pcm.h"
u_int32
PIN_VALIDITY_SET_UNIT(
                u_int32      encoded_value,
                 u_int32      unit_value);
```

Parameters

encoded_value
The encoded value of the start- or end-time details field.

unit_value
The offset unit value to set.

Return Values

Returns the encoded value of the start- or end-time details field set with the unit value passed in.

PIN_VALIDITY_SET_OFFSET

This macro sets the relative offset (number of offset units) in the start- or end-time details value that is passed in.

Syntax

```
#include "pcm.h"
u_int32
PIN_VALIDITY_SET_OFFSET(
               u_int32     encoded_value,
                u_int32          offset_value);
```

Parameters

encoded_value
The encoded value of the start- or end-time details field.

offset_value
The offset value to set.

Return Values

Returns the encoded value of the start- or end-time details field set with the offset value passed in.

PIN_VALIDITY_SET_MODE

This macro sets the relative mode in the start- or end-time details value passed in.

Syntax

```
#include "pcm.h"
u_int32
PIN_VALIDITY_SET_MODE(
                u_int32     encoded_value,
                 pin_validity_modes_t     mode_value);
```

Parameters

encoded_value
The encoded value of the start- or end-time details field.

mode_value
The mode value to set.

Return Values

Returns the encoded value of the start- or end-time details field set with the mode value passed in.

PIN_VALIDITY_DECODE_FIELD

This macro decodes the values of the mode, unit, and offset in the start- or end-time details value passed in and then sets them in mode, unit, and offset variables.

Syntax

```
#include "pcm.h"
void
PIN_VALIDITY_DECODE_FIELD(
                u_int32                   encoded_value,
                pin_validity_modes_t    mode_variable,
                u_int32                   unit_variable,
                 u_int32                   offset_variable);
```

Parameters

encoded_value
The encoded value of the start- or end-time details field.

mode_variable
The mode variable to set.

unit_variable
The unit variable to set.

offset_variable
The offset variable to set.

Return Values

This macro returns nothing.

PIN_VALIDITY_ENCODE_FIELD

This macro takes the mode, unit, and offset values passed in and encodes them into a start- or end-time details field value.

Syntax

```
#include "pcm.h"
u_int32
PIN_VALIDITY_ENCODE_FIELD(
                pin_validity_modes_t    mode_value,
                u_int32                   unit_value,
                 u_int32                   offset_value);
```

Parameters

mode_value
The mode value.

unit_value
The unit value.

offset_value
The offset value.

Return Values

Returns the encoded value of the start- or end-time details field, set with the mode, unit, and offset values passed in.

2

Storable Class Definitions

This chapter provides reference information for Oracle Communications Billing and Revenue Management (BRM) storable class.

For more information about storable class definitions and field definitions, see *BRM Storable Class Reference*.

For information on how to define or modify storable classes and fields, see "Creating, Editing, and Deleting Fields and Storable Classes" in *BRM Developer's Guide*.

For related information, see "Storable Class-to-SQL Mapping" and “About Flists" in *BRM Developer's Guide*.

Fields Common to All Storable Classes

Every BRM storable class requires three fields to create its object in the system. These fields are available to BRM applications and Facilities Modules (FMs) but cannot be written to directly; they are manipulated only by the Storage Manager.

The fields are:

- PIN_FLD_POID. The unique ID for the object.
- PIN_FLD_CREATED_T. The time that the object was created.
- PIN_FLD_MOD_T. The last time the object was modified.

3
Perl Extensions to the PCM Libraries

This chapter contains a list of functions in **pcmif**, the Perl extension to Oracle Communications Billing and Revenue Management (BRM) Portal Communications Module (PCM) library, with links to the description of each function in the library.

For guidelines on using the Perl extensions to create applications, see "Creating Client Applications by Using Perl PCM" in *BRM Developer's Guide*.

For sample Perl scripts using **pcmif**, see "Example Perl Scripts".

Connection Functions

Table 3-1 list the connection function perl extensions to the PCM libraries.

Table 3-1 Connection Functions

Function	Description
pcm_context_close	Closes the given PCM context, disconnects from BRM, and frees memory associated with the context.
pcm_perl_connect	Connects to BRM by using PCM_CONNECT.
pcm_perl_context_open	Opens a PCM context to BRM by using PCM_CONTEXT_OPEN.
pcm_perl_get_session	Obtains the session ID set after login as a printable POID and returns it as a string.
pcm_perl_get_userid	Obtains the user ID set after login as a printable POID and returns it as a string.
pin_perl_time	Returns the time from the **pin_virtual_time** function, which is used to change time in BRM.

Error-Handling Functions

Table 3-2 list the error-handling function perl extensions to the PCM libraries.

Table 3-2 Error-Handling Functions

Function	Description
pcm_perl_destroy_ebuf	Deletes a previously created error buffer from memory.
pcm_perl_ebuf_to_str	Returns a static string with a printable representation of the error buffer.
pcm_perl_is_err	Checks for errors and returns the integer value of the error code in the error buffer.
pcm_perl_new_ebuf	Creates an empty error buffer structure and returns a pointer to it.
pcm_perl_print_ebuf	Runs a **printf** of the printable representation of the error buffer.
pin_set_err	Sets an error buffer.

Flist Conversion Functions

Table 3-3 list the flist conversion function perl extensions to the PCM libraries.

Table 3-3 Flist Conversion Functions

Function	Description
pin_flist_destroy	Deletes an opaque flist.
pin_flist_sort	Sorts the specified flist using PIN_FLIST_SORT.
pin_perl_flist_to_str	Converts an opaque flist into a printable string representation.
pin_perl_str_to_flist	Converts a printable flist into an opaque flist and returns a reference to the flist.

PCM Opcode Functions

Table 3-4 list the PCM opcode function perl extensions to the PCM libraries.

Table 3-4 PCM Opcode Functions

Function	Description
pcm_perl_op	Performs the indicated PCM operation with the given flags and input flist. It returns the resulting flist.

Example Perl Scripts

This section describes sample Perl scripts.

Perl Script Example 1

This sample script performs the following actions:

- It connects to BRM using the login information in the parameters set in the Config section. The **pin.conf** file only needs a dummy user ID entry.
- If there is an argument, it uses that as the POID ID of the data object to read.
- If there is no argument, it uses POID ID 1 as the default.
- It then reads an object with the POID ID using PCM_OP_READ_OBJ and displays the resulting flist.

```
#The first line of the Perl script.
#!/BRM_home/perl/bin/perl
#
#Test a readobj of /data N (defaults to 1).
#Use the following two lines to specify the directory of the pcmif

#files and that you are using the pcmif module.

use lib '.' ;
use pcmif;

# Config section
# Uses pcm_context_open(), so requires pin.conf with userid only

# Set the login information.
 $LOGIN_DB = "0.0.0.1";
 $LOGIN_NAME = "root.0.0.0.1";
 $LOGIN_PASSWD = "password";
```

```
 $CM_HOST = "somehost";

# Setup and connect
# Create an ebuf for error reporting.

  $ebufp = pcmif::pcm_perl_new_ebuf();

# Use a "here" document to assign an flist string to a variable.

  $f1 = <<"XXX"
  0 PIN_FLD_POID POID [0] $LOGIN_DB /service/pcm_client 1 0
  0 PIN_FLD_TYPE               ENUM [0] 1
  0 PIN_FLD_LOGIN           STR [0] "$LOGIN_NAME"
  0 PIN_FLD_PASSWD_CLEAR    STR [0] "$LOGIN_PASSWD"
  0 PIN_FLD_CM_PTR     STR [0] "ip $CM_HOST 11960"
   XXX
   ;

# Use the string-to-flist conversion function to parse the flist string
# that contains the login information and use it to open a PCM #context.

$login_flistp = pcmif::pin_perl_str_to_flist($f1,
                                             $LOGIN_DB, $ebufp);

# Check for errors and print the error report.
          if (pcmif::pcm_perl_is_err($ebufp)) {
              print "flist conversion failed\n";
              pcmif::pcm_perl_print_ebuf($ebufp);
              exit(1);
              }
# Open a PCM context.
$pcm_ctxp = pcmif::pcm_perl_context_open($login_flistp,
                                             $db_no, $ebufp);
# Check for errors and print the status of the action.

  if (pcmif::pcm_perl_is_err($ebufp)) {
      pcmif::pcm_perl_print_ebuf($ebufp);
      exit(1);
     } else {
     $my_session = pcmif::pcm_perl_get_session($pcm_ctxp);
     $my_userid = pcmif::pcm_perl_get_userid($pcm_ctxp);
     print "back from pcmdd_context_open()\n";
     print "    DEFAULT db is: $db_no \n";
     print "    session poid is: ", $my_session, "\n";
     print "    userid poid is: ", $my_userid, "\n";
     }

# See if we should default to 1, or get a number

          if ($#ARGV >= 0) {
                $obj_id = $ARGV[0];
          } else {
                $obj_id = 1;
          }

# Build an flist.
          $f1 = <<"XXX"
          0 PIN_FLD_POID  POID [0] $db_no /data $obj_id 0
          XXX
          ;
# Convert the flist you built from a string to the flist format.
```

```
$flistp = pcmif::pin_perl_str_to_flist($f1, $db_no, $ebufp);

# Check for errors and print the error report.
          if (pcmif::pcm_perl_is_err($ebufp)) {
                print "flist conversion failed\n";
                pcmif::pcm_perl_print_ebuf($ebufp);
                exit(1);
          }
# Convert the flist to a printable string and print it.

          $out = pcmif::pin_perl_flist_to_str($flistp, $ebufp);
          print "IN flist is:\n";
          print $out;

# Perform a PCM operation to read an object and assign the result
# to a variable. Check for errors and print the error report.

$out_flistp = pcmif::pcm_perl_op($pcm_ctxp, "PCM_OP_READ_OBJ", 0,
              $flistp, $ebufp);
          if (pcmif::pcm_perl_is_err($ebufp)) {
                print "robj failed\n";
                pcmif::pcm_perl_print_ebuf($ebufp);
                exit(1);
          }
# Convert the flist for the object you read to a printable string and print it.

$out = pcmif::pin_perl_flist_to_str($out_flistp, $ebufp);
          print "OUT flist is:\n";
          print $out;

# Close the PCM context. Check for errors and print the error report.
          pcmif::pcm_context_close($pcm_ctxp, 0, $ebufp);
          if (pcmif::pcm_perl_is_err($ebufp)) {
                print "BAD close\n",
                    pcmif::pcm_perl_ebuf_to_str($ebufp), "\n";
                exit(1);
          }
          exit(0);
```

Perl Script Example 2

The following example is used to set up an account with a service of type **/service/ip** with the user name **testterm01** (for a test script). It checks for the existence of the service and exits if the service is found. Otherwise, it finds the **/deal** object needed for “IP Basic" (a standard default) and then creates the **/account** and **/service/ip** objects by using PCM_OP_CUST_COMMIT_CUSTOMER.

```
#!/BRM_home/perl/bin/perl

# This is the directory for the pcmif.so and pcmif.pm files.
# For most usage this is not needed, since they will be obtained
# from the default directory (builtin to perl/BRM_home/<vers>/lib).

use lib '.' ;

# The key - You MUST include this to indicate that you are using
# the pcmif extension.

use pcmif;

# The "pcmif::" prefix is a class prefix, meaning that the
```

```
# function "pcm_perl_new_ebuf()" is from the package/class
#"pcmif".
#
# Get an ebuf for error reporting.
#
$ebufp = pcmif::pcm_perl_new_ebuf();


# Do a pcm_connect(), $db_no is a return.

$pcm_ctxp = pcmif::pcm_perl_connect($db_no, $ebufp);


# Convert an ebuf to a printable string.

$ebp1 = pcmif::pcm_perl_ebuf_to_str($ebufp);

# Check for errors. Always do this.

if (pcmif::pcm_perl_is_err($ebufp)) {
pcmif::pcm_perl_print_ebuf($ebufp);
exit(1);
} else {
print "back from pcm_connect()\n";
print "   DEFAULT db is: $db_no \n";
}

# NOTE: The following convention ($DB_NO) was established
# for use with testnap, to substitute the database number
# into a printed flist as it was parsed into testnap.
# We follow the text convention, but we let perl
# do the substitution via this variable (in upper case).
# NOTE: The flist parse should also perform
# this substitution since it gets fed $db_no.
# for testnap convention.
$DB_NO = $db_no;

# Use a "here" document to build an flist string into
# a variable. This flist will then be parsed and
# used in a pcm_op.
#
# search to see if /service/ip "testterm01" is already created

$f1 = <<"XXX"
0 PIN_FLD_POID           POID [0] $DB_NO /search 236 0
0 PIN_FLD_PARAMETERS     STR [0] "ip"
0 PIN_FLD_ARGS      ARRAY [1]
1     PIN_FLD_LOGIN        STR [0] "testterm01"
0 PIN_FLD_RESULTS      ARRAY [0]
1     PIN_FLD_POID    POID [0] 0.0.0.0  0 0
1     PIN_FLD_LOGIN        STR [0] ""
XXX
;
$flistp = pcmif::pin_perl_str_to_flist($f1, $db_no, $ebufp);
if (pcmif::pcm_perl_is_err($ebufp)) {
print "flist conversion to check for testterm01 failed\n";
pcmif::pcm_perl_print_ebuf($ebufp);
exit(1);
}
$out_flistp = pcmif::pcm_perl_op($pcm_ctxp, "PCM_OP_SEARCH", 0, $flistp, $ebufp);
if (pcmif::pcm_perl_is_err($ebufp)) {
print "SEARCH for testterm01 failed\n";
```

```
pcmif::pcm_perl_print_ebuf($ebufp);
exit(1);
}

#
# Check if "testterm01" is there. If it is you do not
# have to recreate.
#
$out = pcmif::pin_perl_flist_to_str($out_flistp, $ebufp);
# XXX warning, no error check

pcmif::pin_flist_destroy($flistp);
pcmif::pin_flist_destroy($out_flistp);

# We converted the output flist into $out above,
# then cleaned the flist objects up.  Now we use
# a perl string matching operator to look for the
# user id we want.
#
if ($out =~ "testterm01") {
print "testterm01 already exists\n" ;
print $out;
exit(0);
}


print "XXX testterm01 does NOT exist\n" ;

#
# First we need the poid of the /deal object - use "IP Basic".
#
$f1 = <<"XXX"
0 PIN_FLD_POID           POID [0] $DB_NO /search 223 0
0 PIN_FLD_ARGS      ARRAY [1]
1     PIN_FLD_NAME        STR [0] "IP Basic"
0 PIN_FLD_RESULTS      ARRAY [0]
1     PIN_FLD_POID    POID [0] 0.0.0.0  0 0
XXX
;
#
$flistp = pcmif::pin_perl_str_to_flist($f1, $db_no, $ebufp);
if (pcmif::pcm_perl_is_err($ebufp)) {
print "flist conversion to search for package failed\n";
pcmif::pcm_perl_print_ebuf($ebufp);
exit(1);
}
$out_flistp = pcmif::pcm_perl_op($pcm_ctxp, "PCM_OP_SEARCH", 0, $flistp, $ebufp);
if (pcmif::pcm_perl_is_err($ebufp)) {
print "SEARCH for package failed\n";
pcmif::pcm_perl_print_ebuf($ebufp);
exit(1);
}

$out = pcmif::pin_perl_flist_to_str($out_flistp, $ebufp);
# XXX warning, no error check


pcmif::pin_flist_destroy($flistp);
pcmif::pin_flist_destroy($out_flistp);

if ($out !~ "/deal") {
print "no package found \n" ;
```

```
print $out;
exit(1);
}

#
# The /deal object poid (which will be <db> /deal <id> <rev>)
# is isolated with index().Then the rest of the line
# (containing the id...) goes into deal_poid, which is
# trimmed by saving the matching pattern
# (ie the id number) and substituting the saved pattern
# (ie just the numbers) for the rest of the line.
#
$deal_at = index($out, "/deal");
$deal_poid = substr($out, $deal_at + 6);
$deal_poid =~ s|([0-9][0-9]*) .*|$1| ;

print "/deal object poid is ", $deal_poid, "\n";

#
# now we fill in an flist for COMMIT_CUSTOMER
#
$f1 = <<"XXX"
0 PIN_FLD_POIDPOID [0] $DB_NO /account 0
0 PIN_FLD_ACCOUNT_OBJPOID [0] $DB_NO /account 0
0 PIN_FLD_AAC_ACCESS STR [0] "setup.fm_term"
0 PIN_FLD_AAC_SOURCE STR [0] "setup.fm_term"
0 PIN_FLD_AAC_VENDOR STR [0] "setup.fm_term"
0 PIN_FLD_AAC_PACKAGE STR [0] "setup.fm_term"
0 PIN_FLD_AAC_PROMO_CODE STR [0] "setup.fm_term"
0 PIN_FLD_AAC_SERIAL_NUM STR [0] "setup.fm_term"
0 PIN_FLD_BILLINFOARRAY [1]
1 PIN_FLD_BILL_TYPEENUM [0] 0
1 PIN_FLD_CURRENCYUINT [0] 840
0 PIN_FLD_PAYINFOARRAY [1]
1 PIN_FLD_NAMEINFO_INDEXUINT [0] 1
0 PIN_FLD_NAMEINFOARRAY [1]
1 PIN_FLD_SALUTATION STR [0] "Mr."
1 PIN_FLD_LAST_NAME STR [0] "testterm01"
1 PIN_FLD_FIRST_NAME STR [0] "testterm01"
1 PIN_FLD_MIDDLE_NAME STR [0] "x"
1 PIN_FLD_TITLE STR [0] "title"
1 PIN_FLD_COMPANY STR [0] "company"
1 PIN_FLD_ADDRESS STR [0] "address"
1 PIN_FLD_CITY  STR [0] "Cupertino"
1 PIN_FLD_STATE         STR [0] "CA"
1 PIN_FLD_ZIP         STR [0] "95014"
1 PIN_FLD_COUNTRY STR [0] "USA"
1 PIN_FLD_EMAIL_ADDR STR [0] "email_addr"
1 PIN_FLD_CONTACT_TYPE STR [0] "contact_type"
0 PIN_FLD_SERVICESARRAY [1]
1 PIN_FLD_SERVICE_OBJPOID [0] $DB_NO /service/ip 0
1 PIN_FLD_LOGIN STR [0] "testterm01"
1 PIN_FLD_PASSWD_CLEAR STR [0] "testterm01"
XXX
;

#
# To avoid quotation problems in the above here document,
# the package is appended via ".".
#
$f1 = $f1 . "1PIN_FLD_DEAL_OBJ POID [0] $DB_NO /deal $deal_poid" ;
```

```
print "flist is now\n";
print $f1;

$flistp = pcmif::pin_perl_str_to_flist($f1, $db_no, $ebufp);
if (pcmif::pcm_perl_is_err($ebufp)) {
pcmif::pcm_perl_print_ebuf($ebufp);
exit(1);
}
$out_flistp = pcmif::pcm_perl_op($pcm_ctxp, "PCM_OP_CUST_COMMIT_CUSTOMER",
0, $flistp, $ebufp);

if (pcmif::pcm_perl_is_err($ebufp)) {
print "BAD op: PCM_OP_CUST_COMMIT_CUSTOMER\n";
pcmif::pcm_perl_print_ebuf($ebufp);
exit(1);
}

$out = pcmif::pin_perl_flist_to_str($out_flistp, $ebufp);
print "OUT flist is \n" ;
print $out;

pcmif::pin_flist_destroy($flistp);
pcmif::pin_flist_destroy($out_flistp);

pcmif::pcm_context_close($pcm_ctxp, 0, $ebufp);
if (pcmif::pcm_perl_is_err($ebufp)) {
print "BAD close\n",
    pcmif::pcm_perl_ebuf_to_str($ebufp), "\n";
exit(1);
```

pcm_context_close

This function closes the given PCM context, disconnects from BRM, and frees memory associated with the context. If a context is no longer needed, make sure you close it.

For more information, see *BRM Opcode Guide*.

Syntax

```
void
pcm_context_close(ctxp, how, ebufp);
```

Parameters

ctxp
A reference to an open PCM context.

how
Defines how to close the connection.
The standard option is to completely close the connection by passing in **0**. However, if you fork a process, make sure that the process which does not make PCM calls any more (usually the child process) closes all open file descriptors (FDs). You can do this by passing **1** as the value of **how**, which is **PCM_CONTEXT_CLOSE_FD_ONLY** in **pcm.h**. This allows the child process (in most cases) to close the FDs without closing the PCM connection in the parent process that spawned it. If you want the child process to continue making PCM calls, open another PCM connection.

ebufp
A reference to an error buffer obtained through pcm_perl_new_ebuf.

Return Values

This function returns nothing.

Error Handling

This function returns any errors to the error buffer.

pcm_perl_connect

This function connects to BRM by using PCM_CONNECT.

Syntax

```
pcm_context_t*
pcm_perl_connect(db_no, ebufp);
```

Parameters

db_no
The variable for the database number.

ebufp
A reference to an error buffer obtained through pcm_perl_new_ebuf.

Return Values

Returns an opaque reference to the PCM context and sets the database number to *db_no* if the function is successful.

Error Handling

This function returns any errors to the error buffer.

pcm_perl_context_open

This function opens a PCM context to BRM by using PCM_CONTEXT_OPEN.

Syntax

```
pcm_context_t*
pcm_perl_context_open(login_flistp, db_no, ebufp);
```

Parameters

login_flistp
A reference to the login flist. The login flist must have a dummy **PIN_FLD_POID**, a valid login type in **PIN_FLD_TYPE**, the **PIN_FLD_LOGIN**, and any other fields required for the given type, usually **PIN_FILD_PASSWD_CLEAR**. Connection Manager (CM) is declared in the **pin.conf** file or by one or more **PIN_FLD_CM_PTR** fields in the login flist.

db_no
The variable for the database number.

ebufp
A reference to an error buffer obtained through pcm_perl_new_ebuf.

Return Values

Returns an opaque reference to the PCM context and sets the database number to *db_no* if the function is successful.

Error Handling

This function returns any errors to the error buffer.

pcm_perl_destroy_ebuf

This function deletes a previously created error buffer from memory.

Syntax

```
void
pcm_perl_destroy_ebuf(ebufp);
```

Parameter

ebufp
A reference to the error buffer to be deleted.

Return Values

This function returns nothing.

Error Handling

This function does not handle errors.

pcm_perl_ebuf_to_str

This function returns a static string with a printable representation of the error buffer.

Syntax

```
char*
pcm_perl_ebuf_to_str(ebufp);
```

Parameter

ebufp
A reference to the error buffer.

Return Values

Returns a static string if the function is successful.

Error Handling

This function returns a null pointer if there are no errors or a printable string if there are errors.

pcm_perl_get_session

This function obtains the session ID set after login as a printable POID and returns it as a string.

Syntax

```
char*
pcm_perl_get_session(ctxp);
```

Parameter

ctxp
A reference to the open PCM context.

Return Values

Returns a printable string containing the session ID if the function is successful.

Error Handling

This function does not handle any errors.

pcm_perl_get_userid

This function obtains the user ID set after login as a printable POID and returns it as a string.

Syntax

```
char*
pcm_perl_get_userid(ctxp);
```

Parameter

ctxp
A reference to the open PCM context.

Return Values

Returns a printable string containing the user ID if the function is successful.

Error Handling

This function does not handle errors.

pcm_perl_is_err

This function checks for errors and returns the integer value of the error code in the error buffer.

Syntax

```
int
pcm_perl_is_err(erbufp);
```

Parameter

erbufp
A reference to the error buffer.

Return Values

Returns **0** if there are no errors. Returns the error code if there are errors.

Error Handling

This function returns the error code if an error occurred.

pcm_perl_new_ebuf

This function creates an empty error buffer structure and returns a pointer to it.

Syntax

```
pin_errbuf_t*
pcm_perl_new_ebuf();
```

Parameters

This function has no parameters.

Return Values

Returns a reference to the error buffer if the function is successful.

pcm_perl_op

This function performs the indicated PCM operation.

Syntax

```
pin_flist_t*
pcm_perl_op(ctxp, op, flag, in_flp, ebufp);
```

Parameters

ctxp
A reference to an open PCM context.

op
The PCM opcode that indicates the operation to be performed. *op* may be a number or symbolic opcode name, as long as it is known to BRM.
For a list of opcode names, see PCM opcode libraries.

flag
A flag for the opcode. See the opcode description for information on the flags each opcode supports. Most opcodes take no flag, which is input as **(int32) 0**.

in_flp
A reference to the input flist.
For the input flist specifications, see PCM opcode libraries.

ebufp
A reference to the error buffer.

Return Values

Returns a reference to the resulting flist if the function is successful. Returns **NULL** if there is a serious error.

> **Note:**
>
> You have to explicitly destroy both the input and return flists. They are not automatically deleted.

Error Handling

This function uses individual-style ebuf error handling. This means the application must explicitly test for an error condition recorded in the error buffer before making other calls to the BRM application programming interface (API).

The following error codes returned from PCM_OP indicate an error in the Portal Communication Protocol (PCP) transmission:

- PIN_ERR_BAD_XDR
- PIN_ERR_STREAM_EOF
- PIN_ERR_STREAM_IO
- PIN_ERR_TRANS_LOST
- PIN_ERR_CM_ADDRESS_LOOKUP_FAILED

> **Note:**
>
> If you see one of these errors, close the context where the error occurred and open a new context. The output flist is undefined, but the input flist is still valid.

pcm_perl_print_ebuf

This function runs a **printf** of the printable representation of the error buffer.

Syntax

```
void
pcm_perl_print_ebuf(ebufp);
```

Parameter

ebufp
A reference to the error buffer to be printed.

Return Values

This function returns nothing.

Error Handling

This function prints the error buffer if there are errors. This function returns **pcm_perl_print_ebufp():NULL ptr** if there are no errors.

pin_flist_destroy

This function deletes an opaque flist.

Syntax

```
void
pin_flist_destroy(flistp);
```

Parameter

flistp
A reference to the flist to delete.

Return Values

This function returns nothing.

Error Handling

This function does not handle errors.

pin_flist_sort

This function sorts the specified flist using PIN_FLIST_SORT.

Syntax

```
void
pin_flist_sort(*flistp, *sort_flistp, reverse, sort_default, ebufp);
```

Parameters

flistp
A reference to the flist being sorted. The flist normally is an array and the sorting is performed on elements of the array. Each element of the array can be a list of fields; it is those fields that get sorted.

sort_listp
A list of fields in each element in *flistp* to use as sort fields. Elements in *flistp* are sorted in this order. If the value of this parameter is **NULL**, **PIN_ERR_BAD_ARG** is returned.

reverse
Reverses the order in which the flist is sorted.

sort_default
Compares nonexistent fields to existing fields.
For detailed information, see "PIN_FLIST_SORT".

ebufp
A reference to the error buffer.

Return Values

This function returns nothing.

Error Handling

This routine uses series-style ebuf error handling. Applications can call any number of series-style ebuf API routines by using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors.

pin_perl_flist_to_str

This function converts an opaque flist into a printable string representation.

For more information, see "PIN_FLIST_TO_STR".

Syntax

```
char*
pin_perl_flist_to_str(flistp, ebufp);
```

Parameters

flistp
A reference to the flist.

ebufp
A reference to the error buffer.

Return Values

Returns the flist in a printable string format if the function is successful. Returns **NULL** if the function fails.

Error Handling

This routine uses series-style ebuf error handling. Applications can call any number of series-style ebuf API routines by using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors.

For more information, see "Understanding API Error Handling and Logging" in *BRM Developer's Guide*.

pin_perl_str_to_flist

This function converts a printable flist into an opaque flist and returns a reference to the flist. If the flist uses the string '$DB_NO' for the database in the POID type fields, the value of *db_no* is substituted. In Perl, it is easier to set a variable $DB_NO and let Perl substitute the "DB_NO" if the flist is defined using **here** documents.

Syntax

```
pin_flist_t*
pin_perl_str_to_flist(str, db_no, ebufp);
```

Parameters

str
A reference to the destination string containing an flist in printable format.

db_no
A reference to the database number. Must be a string containing a BRM database number in dotted decimal format that is used to set the default database for parsing the flist.

ebufp
A reference to the error buffer.

Return Values

Returns the reference to the flist created from the input string if the function is successful. Returns **NULL** if the function fails.

Error Handling

This function uses series-style ebuf error handling. Applications can call any number of series-style ebuf API routines using the same error buffer and check for errors only once at the end of the series of calls. This makes manipulating flists and POIDs much more efficient because the entire logical operation can be completed and then tested once for any errors.

For more information, see "Finding Errors in Your Code" in *BRM Developer's Guide*.

pin_perl_time

This function returns the time from the **pin_virtual_time** function, which is used to change time in BRM. You use this function for testing time-sensitive functions in BRM without affecting the system clock.

For more information, see "pin_virtual_time" in *BRM Developer's Guide*.

Syntax

```
time_t
pin_perl_time();
```

Parameters

This function has no parameters. However, for time offsets to take effect, there must be an entry for **pin_virtual_time** in the **pin.conf** file.

Return Values

Returns the time as a Linux style time value: the number of seconds since 00:00:00 UTC, January 1, 1970.

Error Handling

This function does not handle errors.

pin_set_err

This function sets an error buffer.

Syntax

```
void
pin_set_err(ebufp, location, errclass, pin_err, field, recID, resvd);
```

Parameters

ebufp
A reference to the error buffer to be set.

location
The location of an error, which is one of the PIN_ERRLOC_*xxx*, where *xxx* indicates the subsystem that issued the error.
For details, see "pin_set_err".

errclass
One of the four classes of error PIN_ERRCLASS_*xxx*.
For details, see "pin_set_err".

pin_err
One of the system error messages PIN_ERR_*xxx*.
For details, see "pin_set_err".

field
Set this field to **0** or to the applicable PIN_FLD_*xxx*.

recID
Set this field to **0** or to the record ID of the array element where the error occurred.

resvd
Reserved. Set this field to **0** or to a value chosen to provide further information about the specific error.

Return Values

This function returns nothing.

Error Handling

This function does not handle errors.

4

Storable Class-to-SQL Mapping

This chapter lists each Oracle Communications Billing and Revenue Management (BRM) storable class and the SQL tables to which it is mapped.

Storable Class-to-SQL Mapping

You use SQL directly with the database to generate reports. If you are an experienced system administrator, you can add indexes to improve performance. The default indexes are specified in the **create_indexes.source** file in the *BRM_home***/sys/dm_oracle/data/sql** directory.

> **Caution:**
>
> - Always use the BRM API to manipulate data. Changing data in the database without using the API can corrupt the data.
> - Do not use SQL commands to change data in the database. Always use the API.
> - Do not update or delete the default indexes.

SQL Mapping Matrix

A complete list of SQL tables and fields and their storable-class equivalents is in the file *BRM_home***/sys/dd/data/dd_objects.source**. Indexes are listed in the **create_indexes.source** file in the *BRM_home***/sys/dm_oracle/data/sql** directory.

For storable class-to-SQL mapping information, refer to the storable class descriptions. Each description includes the SQL mapping for every field in the storable class. See "Storable Class Definitions".

SQL Mapping Notes

When looking up SQL mapping indexes, keep in mind the following exceptions.

- The PIN_FLD_INTERNAL_NOTES field in the **/account** storable class is implemented by two fields in two separate tables: the field size is stored in the **/account** storable class as **internal_notes_size**, and the field value is stored in the table **account_internal_notes_buf**.
- The **PIN_FLD_BUFFER** field in the **/data** storable class is implemented by two fields in two separate tables: the field size is stored in the **/data** storable class as **buffer_size**, and the field value (the buffer) is actually stored in the table **data_buffer_buf**.
- SQL **recid** fields correspond to an element ID field.
- All **/event** storable subclasses inherit a set of fields from the **/event** super class, but they are implemented using different tables. The following **/event** storable subclasses are implemented using only the **event_t** table:

  - **/event/activity**
  - **/event/activity/admin**
  - **/event/billing/cycle/arrears**
  - **/event/billing/cycle/fold**
  - **/event/billing/cycle/forward**
  - **/event/billing/debit**
  - **/event/session/pcm_client**

  All other **/event** storable subclasses implemented using the **event_t** table plus one or more additional tables.

- All **/service/*** storable classes inherit a set of fields from the **/service** storable class. In addition, **/service/email** and **/service/pcm_client** are implemented using only the **service_t** table, and **/service/ip** and **/service/admin_client** each require an additional table.
- The **/data** storable class is a general data class that can be used to store any type of data, including blobs. Unless you have specifically created **/data** storable classes, you won't need to access them with SQL since they are generally not used by the system.

Doing SQL Joins

If POIDs (object IDs) are not being used as the join criteria, joins can be done with normal field comparisons.

If object IDs are being used to join tables (for example, to get information about an account and its current balances), simplified join criteria can be used. All tables have either POIDs, which are concatenations of five fields, or they have two-field object IDs, **obj_id0** and **obj_id1**. The **poid_id0** and **poid_id1** fields in the main tables (like **/account**, **/event**, and **/service**) are the same as the **obj_id0** and **obj_id1** fields in their related tables (that are used to implement arrays and substructures), respectively. For example:

```
poid_id0 in account_t = obj_id0 in account_balances_t
poid_id1 in account_t = obj_id1 in account_balances_t
```

The database number (**poid_db**) should be the same for all objects in the same database and you won't need to join on it. In most cases, just joining on the **poid_id0** and **poid_id1** fields are sufficient. The only case where this is not enough is in the case of array elements such as **/event** balance impacts where an SQL **rec_id** (or object element ID) is also required.

The **poid_rev** field is incremented each time an object is modified. This field should not be used or changed. It is not necessary as a join criteria.

rec_id fields are used to match on particular array elements.

Reserved Tables

The following objects/tables listed in Table 4-1 are found in *home***/sys/data/sql/dd_objects.source** file are reserved for BRM use and should not be used by customers:

Table 4-1 Reserved Tables

Object	Reserved SQL Table
/link	link_t
null object	access_table
/who	who_t

SQL Statement Information at Runtime

It is possible to obtain a list of SQL statements which correspond to an operation or sequence of events. See "Increasing the Level of Reporting for a DM" in *BRM System Administrator's Guide* for more details.

5

Sample Applications

This chapter describes the sample programs included with the Oracle Communications Billing and Revenue Management (BRM) SDK, how to use the sample code, and how to run the sample programs.

> **Caution:**
>
> These programs can change or delete data in your BRM database.

About Using the PCM C Sample Programs

BRM SDK includes a set of sample applications and templates using the Portal Communication Model (PCM) C application programming interface (API). You can use these sample programs and templates in the following ways:

- Use the sample programs as code samples for extending BRM components and applications and for writing custom applications.
- Run the corresponding executable application with a sample program to observe the changes it makes in BRM.
- Use the templates, which provide the basic structure for the components, to create your custom components, such as Facilities Modules (FMs) and Data Managers (DMs)

These samples are supported on Linux. Compile these sample programs using the appropriate compiler for your platform.

Finding the PCM C Sample Programs

You can view the sample programs by clicking the links to the sample programs. When you install BRM SDK on Linux, sample programs and templates are found in the following directories:

- Most sample programs and the templates are installed in *BRM_SDK_home***/source/samples** by default.
- Other sample programs can be found in *BRM_SDK_home***/source/samples/apps/c**.
- Templates are located in *BRM_SDK_home***/source/templates**.

You can install BRM SDK on the BRM server at the same time as you install BRM, or independently as an individual component. See "About BRM Install Package" and "Installing Individual BRM Components" in *BRM Installation Guide* for more information.

Description of the PCM C Sample Programs

The sample programs demonstrate how to write code for various tasks when customizing BRM.

Each sample includes these supporting files:

- Source files to view or modify for your own applications.
- Makefiles to compile the sample programs on Linux, if you make changes to the samples.
- A compiled application that verifies that the sample programs work as expected and that allows you to observe the changes the programs make in BRM.
- A **pin.conf** that allows you to specify the information required for the sample application to connect to BRM.

The following tables provide:

- A list of the sample programs and templates.
- A description of each sample program and template.
- Information on any executable program that you can run to observe the results.

Table 5-1 lists a sample for setting makefile macros.

Table 5-1 Setting Makefile Macros (File Located in *BRM_SDK_home*/source/samples)

Sample	Description
env.unix	Shows you how the environment is set up, for example, the location of include directories. The makefiles reference the appropriate environment file for this information. Instructions on setting the makefile macros are included in these text files.

Table 5-2 lists the sample flist files.

Table 5-2 Creating an Flist (Files Located in *BRM_SDK_home*/source/samples/flists/C)

Sample	Description
simple_flist.c	Shows how to create an flist with simple fields. Run **simple_flist.exe** to see a printout of the flist created, which contains a POID and two strings containing the first and last names. For information on how to run **simple_flist**, see "Running the Sample PCM C Programs".
flists_with_arrays.c	Shows how to create flists with arrays containing a single element and multiple elements. Run **flists_with_arrays.exe** to see the flists created by this sample. For information on how to run **flists_with_arrays**, see "Running the Sample PCM C Programs".
flists_with_substructs.c	Shows how to create an flist with a substructure. Run **flists_with_substructs.exe** to see the flists created by this sample. For information on how to run **flists_with_substructs**, see "Running the Sample PCM C Programs".

Table 5-3 lists a sample file for creating a context.

Table 5-3 Creating a Context (File Located in *BRM_SDK_home*/source/samples/context/C)

Sample	Description
create_context.c	Shows you how to open a context, connect to BRM, perform operations, close the context and test if the connection is open. Run **CreateContext.exe** to see how to open a context. For information on how to run **create_context**, see "Running the Sample PCM C Programs".

Table 5-4 lists a sample file for calling an opcode.

Table 5-4 Calling an Opcode (Files Located in BRM_SDK_home/source/samples/callopcode/C)

Sample	Description
test_loopback.c	Shows you how to call an opcode. This sample calls the PCM_OP_TEST_LOOPBACK opcode which just returns the flist that you pass in as the input. Run **test_loopback.exe** to verify that the program returns input flist as the output. For information on how to run **test_loopback**, see "Running the Sample PCM C Programs".

Table 5-5 lists the sample files for client application functions.

Table 5-5 Creating a Client Application (Files Located in BRM_SDK_home/source/samples/apps/c)

Sample	Description
sample_app.c	Shows how to create a customer account with services. For more information about this program, see "Creating Accounts by Using the sample_app.c Program".
sample_del.c	Shows how to remove accounts from BRM. For more information about this program, see "Removing Accounts by Using the sample_del.c Program".
sample_search.c	Shows how to search for objects and fields. For more information about this program, see "Searching by Using the sample_search.c Program".
sample_who.c	Shows how to display the current users. For more information about this program, see "Displaying Current Users by Using the sample_who.c Program".

Table 5-6 lists the FM template files.

Table 5-6 Templates for Creating an FM

Sample	Description
fm_generic_opcode.c	Provides structure for generic (FM) opcodes. See "Using the FM and DM Templates". This file is in *BRM_SDK_home*/**templates/fm_template**.
fm_generic_config.c	Shows you how to map from the opcode to the function. See "Using the FM and DM Templates". This file is in *BRM_SDK_home*/**templates/fm_template**.

Table 5-6 (Cont.) Templates for Creating an FM

Sample	Description
op_define.h	Header file required by FM templates which defines **PCM_OP_GENERIC**. This file is in *BRM_SDK_home***/templates/fm_template**.

Table 5-7 lists the template file for creating a DM.

Table 5-7 Template for Creating a DM

Sample	Description
dm_generic.c	Shows the basic structure of a Data Manager. See "Using the FM and DM Templates". This file is in *BRM_SDK_home***/templates/dm_template**.

Table 5-8 lists the sample files for using the multithreaded application (MTA) APIs.

Table 5-8 Using the Multithreaded Application (MTA) API

Sample	Description
pin_mta_monitor.c (located in *BRM_SDK_home***/bin**)	Sample monitoring utility.
pin_mta_test.c (located in *BRM_SDK_home***/source/samples/apps/c/mta_sample**)	Sample test program using the MTA framework.

Compiling the Sample PCM C Programs

In addition to using the sample programs as a working programming example, you can also use them as a basis for your own applications. You can make changes to the sample programs, compile, and run them to test your changes. The sample programs directory includes the following files:

- **env.unix** to set the environment
- Makefiles for Linux to compile the samples

To compile the sample programs on Linux:

1. Go to *BRM_SDK_home***/source/samples** and open **env.nt** or **env.unix**, depending on your operating system.
2. Set up the path for the environment by following the instructions in the file.
3. Save the file.
4. Compile using the appropriate **make** utility:

```
make
```

Running the Sample PCM C Programs

The executable versions of the sample programs are provided in addition to the source files. To see the output generated by a sample program, follow these basic steps:

1. Go to the directory where the sample program is located. The default structure is: *BRM_SDK_home***/source/samples** or *BRM_SDK_home***/source/samples/apps/c**.
2. Edit the entry in the configuration file **pin.conf** to point to the Connection Manager (CM).
3. Run the program by running the executable file, for example:

```
create_context.exe
```

> **Note:**
>
> Some sample programs require parameters or have special syntax requirements. For more information, see "Creating Accounts by Using the sample_app.c Program", "Removing Accounts by Using the sample_del.c Program", or "Searching by Using the sample_search.c Program".

Using the FM and DM Templates

In addition to the sample programs, the BRM SDK includes FM and DM templates that you can use as starting points for your own customized versions. You can make changes to the templates, compile them, and run them to test your changes. Makefiles and .dlls are provided for the templates in *BRM_SDK_home***/source/templates/fm_template** and *BRM_SDK_home***/source/templates/dm_template**.

The templates are provided in two forms:

- C files that you can modify and compile according to the instructions in Compiling the Sample PCM C Programs.
- DSP files that you can open as projects in Microsoft Visual Studio.

See "Testing New or Customized Policy FMs" and "Testing New or Customized DMs" in *BRM Developer's Guide* for information about testing the modified templates.

Creating Accounts by Using the sample_app.c Program

The **sample_app.c** program creates an account with services in the specified package. You can modify this program to add new services to an account or to create dummy accounts to test BRM functionality.

This program performs the following actions:

1. Opens a database channel
2. Retrieves the specified package
3. Adds the customer information to the package
4. Creates the customer account
5. Closes the database channel

For information on the structure and parameters, see the source file **sample_app.c** located in *BRM_SDK_home***/source/samples/apps/c**.

Syntax for sample_app.c

Run the program with appropriate options listed in Table 5-9 and package name. The options can be in any order except that the name of the package must be the last entry.

```
% sample_app  [-l login] [-p password] <package>
```

Table 5-9 sample_app.c Account Creation Parameters

Parameter	Description	Condition
-l	Login	Required
-p	Password	Required
-d	Set error level	Optional
-h	Print standard error	Optional

The following example accepts the account logon and password for **jsmith**.

```
sample_app -l jsmith -p my_password email_package
```

Removing Accounts by Using the sample_del.c Program

The **sample_del.c** program finds an account by searching for one of its service logins, and then deletes the account and all of its related objects.

> **Caution:**
>
> This program deletes accounts permanently. You cannot retrieve any accounts that you delete by running this program.

For information on the structure and parameters, see the source file, **sample_del.c** located in *BRM_SDK_home***/source/samples/apps/c**.

Syntax for sample_del.c

The **sample_del.c** program does not take any parameters.

```
% sample_del  /servicetype login
```

This example deletes the **/service/ip** account with the login **smith**:

```
% sample_del /service/ip smith
```

Searching by Using the sample_search.c Program

The **sample_search.c** program demonstrates the different types of searches in BRM.

- Read-object search with single result expected

Searches for the primary account object and displays the results with PIN_FLIST_PRINT.

- Read-fields search with multiple results expected

  Searches for the POID, merchant, and status of all nonbillable accounts in the database.

- Step search

  Searches for services that require AES-encrypted passwords. The first 10 such services are retrieved in 2 blocks of 5 services each.

For information on the structure, see the source file **sample_search.c** located in *BRM_SDK_home***/source/samples/apps/c**.

Syntax for sample_search.c

The **sample_search.c** program does not take any parameters.

```
% sample_search
```

Displaying Current Users by Using the sample_who.c Program

The **sample_who.c** program finds all the active dialup sessions in the database, looks up the login for each user with an open session, and displays a list of all customers currently logged in to your Internet service.

For information on the structure, see the source file **sample_who.c** located in *BRM_SDK_home***/source/samples/apps/c**.

Syntax for sample_who.c

The **sample_who.c** program does not take any parameters.

```
% sample_who
```

Troubleshooting the sample_app.c Application

If you cannot run the **sample_app** application, use this information to identify any problems and resolve them.

Problem: Test Failed

```
sample# sample_app
bad/no "userid" from pin.conf file

Test Failed, See Log File.
```

Solution

Edit the **sample_app** configuration file to include the correct **userid** entry and make sure the application is configured correctly.

Problem: Bad Port Number

```
sample# sample_app
(11400): bad receive of login response, err 4
(11400): login failed 4

Test Failed, See Log File
```

```
sample# cat default.pinlog
E Fri Mar 15 14:56:44 1998  db2.corp <no name>:11393  pcm.c(1.41):90
         Connect open failed (4/100) in pcm_context_open
E Fri Mar 15 14:58:39 1998  db2.corp  <no name>:11400  pcm.c(1.41):90
         Connect open failed (4/5) in pcm_context_open
```

Solution

Edit the **cm_ptr** entry in the **sample_app** configuration file with the valid CM port number.

Problem: Customer Account Creation Error

```
sample# sample_app

Test Failed, See Log File

E Fri Mar 15 15.10:37 1998  db2.corp  :11405  sample_app.c:167
         op_cust_create_acct error [location= class= errno= field num= recid=<0>
reserved=<0>]
```

Solution

Load the BRM objects into the database.

About Using the PCM C++ Sample Programs

BRM SDK includes a set of sample applications using the PCM C++ API. You can use these sample programs in the following ways:

- Use the sample programs as code samples for extending BRM components and applications and for writing custom applications.
- Run the corresponding executable application with a sample program to observe the changes it makes in BRM.

These samples are supported Linux. Compile these sample programs using the appropriate compiler for your platform.

Finding the Sample PCM C++ Programs

When you install BRM SDK on Linux, the sample programs are installed by default in *BRM_home*/**InfranetSDK/source/samples**.

You can also display the sample programs by clicking the links in this document.

> **Note:**
>
> The installation directory is called *BRM_SDK_home* in the documentation.

You can install BRM SDK on the BRM server at the same time as you install BRM, or independently as an individual component. See "About BRM Install Package" and "Installing Individual BRM Components" in *BRM Installation Guide* for more information.

Description of the Sample PCM C++ Programs

The sample programs demonstrate how to write code for various tasks when customizing BRM.

Each sample includes these supporting files:

- Source files to view or modify for your own applications
- Makefiles to compile the sample programs on Linux, if you make changes to the samples
- A compiled application that verifies that the sample programs work as expected and that allows you to observe the changes the programs make in BRM
- A configuration file **pin.conf** that allows you to specify the information required for the sample application to connect to BRM

The following tables provide:

- A list of the sample programs
- A description of each sample program
- Information on any executable program that you can run to observe the results

Table 5-10 lists the file for setting makefile macros.

Table 5-10 Setting Makefile Macros (File Located in BRM_SDK_home/source/samples)

Sample	Description
env.unix	Shows you how the environment is set up, for example, the location of include directories. The makefiles reference the appropriate environment file for this information. Instructions on setting the makefile macros are included in these text files.

Table 5-11 lists the sample files for creating an flist.

Table 5-11 Creating an Flist (Files Located in BRM_SDK_home/source/samples/flists/C++)

Sample	Description
simple_flist.cpp	Shows how to create an flist with simple fields. Run **simple_flist.exe** to see a printout of the flist created, which contains a POID and two strings containing the first and last names. For information on how to run **simple_flist**, see "Running the Sample PCM C Programs".
flists_with_arrays.cpp	Shows how to create flists with arrays containing a single element and multiple elements. Run **flists_with_arrays.exe** to see the flists created by this sample. For information on how to run **flists_with_arrays**, see "Running the Sample PCM C Programs".
flists_with_substruct.cpp	Shows how to create an flist with a substructure. Run **flists_with_substruct.exe** to see the flists created by this sample. For information on how to run **flists_with_substruct**, see "Running the Sample PCM C Programs".

Table 5-12 lists the sample file for creating a context.

Table 5-12 Creating a Context (File Located in BRM_SDK_home/source/samples/context/C++)

Sample	Description
create_context.cpp	Shows you how to open a context, connect to BRM, perform operations, test if the connection is open, and close the context. Run **create_context.exe** to verify that the program returns input flist as the output. For information on how to run **create_context**, see "Running the Sample PCM C Programs".

Table 5-13 lists the sample file for calling an opcode.

Table 5-13 Calling an opcode (File Located in BRM_SDK_home/source/samples/callopcode/C++)

Sample	Description
test_loopback.cpp	Shows you how to call an opcode. This sample calls the PCM_OP_TEST_LOOPBACK opcode which just returns the flist that you pass in as the input. Run **test_loopback.exe** to verify that the program returns input flist as the output. For information on how to run **test_loopback**, see "Running the Sample PCM C Programs".

Table 5-14 lists the sample files for creating a client application.

Table 5-14 Creating a Client Application (Files Located in BRM_SDK_home/source/samples/apps/C++)

Sample	Description
sample_PinBD.cpp	Shows how to use the class PinBigDecimal. This program illustrates how to create a big decimal number from a string or double, the use of various rounding modes and setting the number of decimal places, the use of mathematical functions, etc. Run **sample_PinBD.exe** to see how the program works. For information on how to run **sample_PinBD**, see "Running the Sample PCM C Programs".

Table 5-15 lists the sample files for using the multithreaded application (MTA) APIs.

Table 5-15 Using the Multithreaded Application (MTA) API

Sample	Description
pin_mta_monitor (located in *BRM_SDK_home/bin*)	Sample monitoring utility.
pin_mta_test.c (located in *BRM_SDK_home/source/samples/apps/c/mta_sample*)	Sample test program using the MTA framework.

Compiling the Sample PCM C++ Programs

In addition to using the sample programs as working programming examples, you can also use them as a basis for your own applications. You can make changes to the sample programs,

compile, and run them to test your changes. The sample programs directory includes the following files:

- **env.unix** to set the environment
- Makefiles for Linux to compile the samples

To compile the sample programs:

1. Go to *BRM_SDK_home***/source/samples**, and open **env.unix**.
2. Set up the path for the environment by following the instructions in the file.
3. Save the file.
4. Compile using the **make** utility:

   ```
   make
   ```

Running the Sample PCM C++ Programs

The executable versions of the sample programs are provided. To see the output generated by a sample program, follow these basic steps:

1. Go to the directory where the sample program is located. The default path is *BRM_SDK_home***/source/samples**.
2. Edit the entry in the configuration file **pin.conf** to point to the CM.
3. Run the program by running the executable, for example:

   ```
   create_context.exe
   ```

About Using the PCM Java Sample Programs

BRM SDK includes a set of sample applications using the PCM Java API. You can use these sample programs in the following ways:

- Use the sample programs as code samples for extending BRM components and applications and for writing custom applications.
- Run the corresponding executable application with a sample program to observe the changes it makes in BRM.

These samples are supported on Linux. Compile these sample programs using the appropriate compiler for your platform.

Finding the Sample PCM Java Programs

When you install BRM SDK, the sample programs are installed by default in *BRM_home***/ InfranetSDK/source/samples.**

You can also display the sample programs by clicking the links in this document.

> **Note:**
>
> The installation directory is called *BRM_SDK_home* in the documentation.
>
> You can install BRM SDK on the BRM server at the same time as you install BRM, or independently as an individual component. See "About BRM Install Package" and "Installing Individual BRM Components" in *BRM Installation Guide* for more information.

Description of the Sample PCM Java Programs

The sample programs demonstrate how to write code for various tasks when customizing BRM.

Each sample includes these supporting files:

- Source files to view or modify for your own applications
- Makefiles to compile the sample programs, if you make changes to the samples
- A compiled application that verifies that the sample programs work as expected and that allows you to observe the changes the programs make in BRM
- A configuration file **infranet.properties** that allows you to specify the information required for the sample application to connect to BRM

The following tables provide:

- A list of the sample programs and makefiles
- A description of each sample program and makefile
- Information on any executable program that you can run to observe the results

Table 5-16 lists the sample file for setting the makefile macros.

Table 5-16 Setting Makefile Macros (File Located in BRM_SDK_home/source/samples)

Sample	Description
env.unix	Shows you how the environment is set up, for example, the location of include directories. The makefiles reference the appropriate environment file for this information. Instructions on setting the makefile macros are included in these text files.

Table 5-17 lists the sample files for creating an flist.

Table 5-17 Creating an Flist (Files Located in BRM_SDK_home/source/samples/flists/Java)

Sample	Description
SimpleFlist.java	Shows how to create an flist with simple fields. Run **SimpleFlist.class** to see a printout of the flist created, which contains a POID and two strings containing the first and last names. For information on how to run **SimpleFlist**, see "Running the Sample PCM C Programs".

Table 5-17 (Cont.) Creating an Flist (Files Located in BRM_SDK_home/source/samples/flists/Java)

Sample	Description
FlistsWithArrays.java	Shows how to create flists with arrays containing a single element and with arrays containing multiple elements. Run **FlistsWithArrays.class** to see the flists created by this sample. For information on how to run **FlistsWithArrays**, see "Running the Sample PCM C Programs".
FlistsWithSubstructs.java	Shows how to create an flist with a substructure. Run **FlistsWithSubstructs.class** to see the flists created by this sample. For information on how to run **FlistsWithSubstructs**, see "Running the Sample PCM C Programs".

Table 5-18 lists the sample file for creating a context.

Table 5-18 Creating a Context (File Located in BRM_SDK_home/source/samples/context/Java)

Sample	Description
CreateContext.java	Shows you how to open a context, connect to BRM, perform operations, test if the connection is open, and close the context. Run **CreateContext.class** to see how to open a context. For information on how to run **CreateContext**, see "Running the Sample PCM C Programs".

Table 5-19 lists the sample file for calling an opcode.

Table 5-19 Calling an Opcode (File Located in BRM_SDK_home/source/samples/callopcode/Java)

Sample	Description
TestLoopback.java	Shows you how to call an opcode. This sample calls the PCM_OP_TEST_LOOPBACK opcode which just returns the flist that you pass in as the input. Run **TestLoopback.class** to verify that the program returns input flist as the output. For information on how to run **TestLoopback**, see "Running the Sample PCM C Programs".

Table 5-20 lists the sample files for creating a client application.

Table 5-20 Creating a Client Application (Files Located in BRM_SDK_home/source/samples/apps/Java)

Sample	Description
CreateCustomUsageEvent.java	Shows you how to generate an email activity event for a particular account. Run **CreateCustomUsageEvent.class** to see how the program works. For more information on **CreateCustomUsageEvent**, see "Creating Events by Using the CreateCustomUsageEvent.java Program" For information on how to run **CreateCustomUsageEvent**, see "Creating Events by Using the CreateCustomUsageEvent.java Program".

Table 5-20 (Cont.) Creating a Client Application (Files Located in BRM_SDK_home/source/samples/apps/Java)

Sample	Description
CreateCustomer.java	Shows you how to create a new customer through the user interface defined in **CreateCustomerUI.java,**using the account information definition from **CreateCustomerAccountInfo.java** and the model created by **CreateCustomerModel.java** Run **CreateCustomer.class** to see how to create a customer using these four programs. For more information on **CreateCustomer**, see "Creating Accounts by Using the CreateCustomer.java Program" For information on how to run **CreateCustomer**, see "Running the Sample PCM C Programs".
CreateCustomerUI.java	Defines the user interface used by **CreateCustomer**.
CreateCustomerAccountInfo.java	Defines the account information and holds the data.
CreateCustomerModel.java	Shows you how to create new customers by creating flists to pass information to it, including customer name and address, pertinent package, billing information, invoice data, and so on. Then it adds the requested login and password to each service array element and creates the customer in the BRM database. Of the four **CreateCustomer** programs, **Create CustomerModel.java** is where all the BRM actions take place in this program.

Compiling the Sample PCM Java Programs

In addition to using the sample programs as working programming examples, you can also use them as a basis for your own applications. You can make changes to the sample programs, compile, and run them to test your changes. The sample programs directory includes the following files:

- **env.unix** to set the environment
- Makefiles to compile the samples

To compile the sample programs:

> **Note:**
>
> To compile the sample programs, you must have a Java compiler installed on your system. For a list of compatible versions of Java, see "Additional BRM Software Requirements" in *BRM Compatibility Matrix*.

1. Go to *BRM_SDK_home***/source/samples**, and open **env.unix**.
2. Set up the path for the environment by following the instructions in the file. Make sure the JDK_HOME variable includes the absolute path of your Java compiler.
3. Save the file.
4. Compile using the **make** utility:

```
make
```

Running the Sample PCM Java Programs

The executable versions of the sample programs are provided. To see the output generated by a sample program, follow these basic steps:

1. Go to the directory where the sample program is located. The default structure is: *BRM_SDK_home***/source/samples**.
2. Edit the configuration file **infranet.properties** to point to the CM.
3. Set the classpath to:

   ```
   java -classpath <path to jar files> <sample_name>
   ```

   For example:

   ```
   classpath/BRM_SDK_home/jars/pcm.jar;/BRM_SDK_home/jars/pcmext.jar;. SimpleFlist
   ```

4. Run the program, for example:

   ```
   java create_context
   ```

Creating Accounts by Using the CreateCustomer.java Program

The **CreateCustomer.java** program creates an account with services in the specified package. You can modify this program to add new services to an account or to create dummy accounts to test BRM functionality.

This program performs the following actions:

1. Opens a database channel
2. Retrieves the specified package
3. Adds the customer information to the package
4. Creates the customer account
5. Closes the database channel

For information on the structure and parameters, look at the source file **CreateCustomer.java** located in *BRM_SDK_home***/source/samples/apps/Java**.

Creating Events by Using the CreateCustomUsageEvent.java Program

The **CreateCustomUsageEvent.java** program simulates customer activity by creating an activity event for an email service object. Use this program to generate any number of email events.

For information on the structure, see the source file **CreateCustomUsageEvent.java** located in *BRM_SDK_home***/source/samples/apps/Java**.

Running the CreateCustomUsageEvent Program

1. Create the storable class of type **event/activity/email** and these custom fields.

   ```
   EMAIL_EVENT_INFO   PIN_FLDT_SUBSTRUCT [0]   ID# 10001
   EMAIL_FROM    PIN_FLDT_STR [0]             10002
   ```

```
EMAIL_TO      PIN_FLDT_STR [0]            10003
```

For information, see "Creating, Editing, and Deleting Fields and Storable Classes" in *BRM Developer's Guide*.

2. Follow the instructions in "Making Custom Fields Available to Your Applications" in *BRM Developer's Guide* to make the custom fields available to your applications.
3. Restart the CM, the client tools, and other components.
4. Run **CreateCustomUsageEvent** to generate email activity events:

```
java CreateCustomUsageEvent
```

About Using the PCM Perl Sample Programs

BRM SDK includes a set of sample applications using the PCM Perl API. You can use these sample programs in the following ways:

- Use the sample programs as code samples for extending BRM components and applications and for writing custom applications.
- Run the corresponding executable application with a sample program to observe the changes it makes in BRM.

These samples are supported on Linux. Compile these sample programs using the appropriate compiler for your platform.

Finding the Sample PCM Perl Programs

When you install BRM SDK on Linux, the sample programs are installed by default in *BRM_home*/**InfranetSDK/source/samples**.

You can also display the sample programs by clicking the links in this document.

> **Note:**
>
> The installation directory is called *BRM_SDK_home* in the documentation.

You can install BRM SDK on the BRM server at the same time as you install BRM, or independently as an individual component. See "About BRM Install Package" and "Installing Individual BRM Components" in *BRM Installation Guide* for more information.

Description of the Sample PCM Perl Programs

The sample programs demonstrate how to write code for various tasks when customizing BRM.

Each sample includes these supporting files:

- Source files to view or modify for your own applications
- A compiled application that you can run to verify that the sample programs work as expected and to observe the changes the program makes in BRM

- A configuration file **pin.conf** where you specify the configuration information for the sample application to connect to BRM

The following tables provide:

- A list of the sample programs
- A description of each sample program
- Information on any executable program that you can run to observe the results

Table 5-21 lists the sample files for creating an flist.

Table 5-21 Creating an Flist (Files Located in *BRM_SDK_home*/source/samples/flists/perl)

Sample	Description
simple_flist.pl	Shows how to create an flist with simple fields. Run simple_flist.pl to see a printout of the flist created, which contains a POID and two strings containing the first and last names.
flist_with_arrays.pl	Shows how to create flists with arrays containing a single element. Run **flist_with_arrays.pl** to see the flist created by this sample.
flist_with_substruct.pl	Shows how to create an flist with a substructure. Run **flist_with_substruct.pl** to see the flist created by this sample

Table 5-22 lists the sample files for creating a context.

Table 5-22 Creating a Context (Files Located in BRM_SDK_home/source/samples/context/perl)

Sample	Description
connect.pl	Shows you how to open a context, connect to BRM using **pin.conf** parameters, perform operations, test if the connection is open, and close the context. Run **contect.pl** to verify that the program returns input flist as the output. For information on how to run **connect.pl**, see "Running the Sample PCM C Programs".
create_context.pl	Shows you how to open a context, connect to BRM using logon information within the program, perform operations, test if the connection is open, and close the context. Run **create_context.pl** to demonstrate how to open a context. For information on how to run **create_context.pl**, see "Running the Sample PCM C Programs".

Table 5-23 lists the sample file for calling an opcode.

Table 5-23 Calling an Opcode (Files Located in BRM_SDK_home/source/samples/callopcode/perl)

Sample	Description
test_loopback.pl	Shows you how to call an opcode. This sample calls the PCM_OP_TEST_LOOPBACK opcode which just returns the flist that you pass in as the input. Run **test_loopback.pl** to verify that the program returns input flist as the output. For information on how to run **test_loopback.pl**, see "Running the Sample PCM C Programs".

Running the Sample PCM Perl Programs

The executable versions of the sample programs are provided. To see the output generated by a sample program, follow these basic steps:

1. Go to the directory where the sample program is located. The default structure is: *BRM_home*/**InfranetSDK/source/samples**.
2. Edit the entry in the configuration file **pin.conf** to point to the CM.
3. Run the program by executing the program name under Perl, for example:

```
perl create_context.pl
```

> **Note:**
>
> Use the Perl installed by the SDK (or with the BRM server), located in *BRM_home*/**perl/bin/perl**. This version of Perl is preconfigured for BRM.