Dynamic Probe Class Library (DPCL): Class Reference Manual

Version 0.3

March 25, 1999

Dr. Douglas M. Pase

email:pase@us.ibm.com

IBM Corporation

RS/6000 Development

522 South Road, MS P-963

Poughkeepsie, New York 12601

Draft Document

Table of Contents

1.0 Function Group AisHandler

1.1 Supporting Data Types

1.1.1 AisHandlerType

Synopsis

```
#include <AisHandler.h>

typedef int (*AisHandlerType)(int fd_or_sig)
```

Description

This data type represents a function pointer that points to an event handler that is called when a noteworthy event takes place. Noteworthy events occur when a file descriptor managed by the instrumentation system receives input, clears space for output, or a signal managed by the instrumentation system has been raised.

The function returns an integer value with the following meaning. If the mechanism that generated the event is a file descriptor and the file descriptor reaches an end-of-file condition, the handler function is to return a value of -1. This indicates to the system that the file descriptor is to be closed and removed from the list of watched file descriptors. If the returned value is any other value than -1 or the event that occured was a signal, the return value is ignored.

1.2 Ais_add_fd

Synopsis

```
#include <AisHandler.h>
AisStatus Ais_add_fd(int fd, AisHandlerType handler)
```

Parameters

`fd`	file descriptor
`handler`	function handler for this socket

Description

Add a file descriptor and input handler to the list of file descriptors managed by the instrumentation system. When input is received by the file descriptor, the handler is called to handle the input. The handler is expected to accept the file descriptor as its input parameter.

Return value

`ASC_success`	request successful
`ASC_operation_failed`	request failed

See Also

`Ais_add_signal, Ais_next_fd, Ais_remove_fd, Ais_remove_signal`

1.3 Ais_add_signal

Synopsis

```
#include <AisHandler.h>

AisStatus Ais_add_signal(int signal, AisHandlerType handler)
```

Parameters

`signal`	signal to be caught
`handler`	function handler for this signal

Description

Add a signal and signal handler to the list of signals managed by the instrumentation system. When a signal is received, the handler is called to handle the signal. The handler is expected to accept the signal as its input parameter. The instrumentation system ensures that signals registered with the instrumentation system will not interfere with its system calls. Signal handlers executed by the instrumentation system are executed on the normal application stack. In the event that multiple signals occur while a signal handler is being executed, the executing handler is completed before the next handler is begun. This provides a measure of safety for operations that are normally considered unsafe for signal handlers, such as memory allocation.

Return value

`ASC_success`	request successful
`ASC_duplicate_signal`	attempt to add a handler for a signal that already has a handler
`ASC_invalid_operand`	attempt to add a handler for a signal which does not exist
`ASC_operation_failed`	system call to add a signal failed

See Also

`Ais_add_fd, Ais_next_fd, Ais_remove_fd, Ais_remove_signal`

1.4 Ais_next_fd

Synopsis

```
#include <AisHandler.h>
void Ais_next_fd(int &fd_or_sig, AisHandlerType &handler)
```

Parameters

`fd_or_sig`	file descriptor or signal number
`handler`	file descriptor or signal handler function

Description

Return the file descriptor or signal number and associated handler of the next event to occur.

See Also

`Ais_add_fd, Ais_add_signal, Ais_remove_fd, Ais_remove_signal`

1.5 Ais_override_default_callback

Synopsis

```
#include <AisHandler.h>
```

```
AisStatus Ais_override_default_callback(unsigned msg_type,
GCBFuncType fp_arg, GCBTagType tag_arg, GCBFuncType *prev_fp,
GCBTagType *prev_tag)
```

Parameters

`msg_type`	message key
`fp_arg`	new callback function
`tag_arg`	new callback tag
`prev_fp`	previous callback function
`prev_tag`	previous callback tag

Description

Replace the system callback associated with an event and replace with a new, user-specified callback. Candidate events for replacement in this fashion are AIS_EXIT_MSG, AIS_ERROR_MSG, and AIS_DEFAULT_CB. The callback and tag values that were associated with the specified event are returned to the user.

Return value

`ASC_success`	request successful
`ASC_operation_failed`	request failed

See Also

1.6 Ais_remove_fd

Synopsis

```
#include <AisHandler.h>
AisStatus Ais_remove_fd(int fd)
```

Parameters

`fd`	file descriptor

Description

Remove a file descriptor from the list of descriptors the instrumentation system manages. The file descriptor is unaffected by this operation, that is, it is neither closed nor flushed.

Return value

`ASC_success`	request successful
`ASC_operation_failed`	request failed

See Also

`Ais_add_fd, Ais_add_signal, Ais_remove_fd, Ais_remove_signal`

1.7 Ais_query_signal

Synopsis

```
#include <AisHandler.h>

AisHandlerType Ais_query_signal(int signal)
```

Parameters

`signal`	signal for which handling is to be removed

Description

This function returns a pointer to the signal handler function for the specified signal, or 0 if there is none.

Return value

A pointer to the signal handler function for the specified signal if there is one. Otherwise 0 if there is no handler or the signal parameter does not represent a valid signal.

See Also

`Ais_add_fd, Ais_add_signal, Ais_next_fd, Ais_remove_fd`

1.8 Ais_remove_signal

Synopsis

```
#include <AisHandler.h>
AisStatus Ais_remove_signal(int signal)
```

Parameters

`signal`	signal for which handling is to be removed

Description

Remove a signal and signal handler from the list of signals the instrumentation system manages. A previous handler is *not* restored for this signal.

Return value

`ASC_success`	signal handler was successfully removed, or there was no handler to be removed
`ASC_invalid_operand`	attempt to remove a handler for a signal that does not exist
`ASC_operation_failed`	system call to delete a signal failed

See Also

`Ais_add_fd, Ais_add_signal, Ais_next_fd, Ais_remove_fd`

2.0 class AisStatus

2.1 Supporting Data Types

2.1.1 AisStatusCode

Synopsis

```
#include <AisStatus.h>
AisStatusCode {
    ASC_success,                 // success
    ASC_failure,                 // failure
    ASC_insufficient_memory,     // insufficient memory
    ASC_invalid_expression,      // invalid expression
    ASC_invalid_value_ref,       // invalid value reference
    ASC_invalid_internal_tree,   // invalid internal tree
    ASC_invalid_src_code_tree,   // invalid source code tree
    ASC_invalid_constructor,     // invalid constructor
    ASC_invalid_operator,        // invalid operator
    ASC_invalid_operand,         // invalid operand
    ASC_operation_failed,        // operation failed
    ASC_empty_object,            // empty object
    ASC_actual_point_mismatch,   // probe containing one-shot
installed at
                                 // wrong point
    ASC_contains_actual,         // one-shot must not contain
actual param
    ASC_contains_data_ref,       // one-shot must not contain
data refs
    ASC_unknown_status,          // unknown status
                                 //
    ASC_internal_error,          // internal error
                                 //
```

```
    ASC_exist_pid,              // exist pid
    ASC_invalid_pid,            // invalid pid
    ASC_terminated_pid,         // terminated pid
    ASC_no_procsinfo,           // no procsinfo available
    ASC_not_runnable_pid,       // not a runnable pid
    ASC_authorization_failed,   // authorization failed
    ASC_dead_code,              // dead code
    ASC_duplicate_signal,       // duplicate signal
    ASC_signal_not_found,       // signal not found
    ASC_null_pointer,           // null pointer
    ASC_install_failed,         // install failed
    ASC_remove_failed,          // remove failed
    ASC_activate_failed,        // activate failed
    ASC_deactivate_failed,      // deactivate failed
    ASC_communication_failure,  // communication failure
    ASC_uninitialized_process,  // uninitialized process
    ASC_uninitialized_daemon,   // uninitialized daemon
    ASC_non_positive_value,     // non-positive value
    ASC_missing_phase_str,      // missing Phase_Str object
    ASC_remove_msgh_failed,     // remove message handle failed
    ASC_missing_pmod,           // missing PModEntry object
    ASC_missing_bp_func,        // missing BPatch_function
object
    ASC_create_msgh_failed,     // create message handle failed
    ASC_missing_predef_func,    // missing predefined function
    ASC_create_phase_failed,    // create PhaseEntry failed
    ASC_missing_phase,          // missing PhaseEntry object
    ASC_phase_exit_done,        // PhaseEntry's exit funcs are
done
    ASC_bad_processd,           // bad ProcessD object
```

```
    ASC_invalid_phase,          // invalid phase
    ASC_duplicate_shm_init,     // duplicate shm init
    ASC_shm_init_failed,        // shm init failed
    ASC_shmat_failed,           // shmat failed
    ASC_shmget_failed,          // shmget failed
    ASC_shmdt_failed,           // shmdt failed
    ASC_shmctl_failed,          // shmctl failed
    ASC_shm_object_alloc_failed,// shm object alloc failed
    ASC_shm_block_alloc_failed, // shm block alloc failed
    ASC_mismatch_pid,           // mismatched pid
    ASC_shm_attach_failed,      // shm attach failed
    ASC_msg_init_failed,        // "msg init failed
    ASC_msg_read_failed,        // msg read failed
    ASC_shm_verify_failed,      // shm verify failed
    ASC_invalid_client,         // invalid client
    ASC_duplicate_phase,        // duplicate phase
    ASC_irpc_failed,            // IRPC failed
    ASC_phase_null_data_func,   // phase has a null data_func
    ASC_phase_malloc_data_failed, // malloc data for a phase
failed

//  process related
    ASC_missing_aout,           // missing a.out file
    ASC_destroyed_process,      // process has been destroyed
    ASC_disconnecting_process,  // process is disconnecting
    ASC_duplicate_create,       // duplicate create
    ASC_duplicate_connect,      // duplicate connect
    ASC_duplicate_attach,       // duplicate attach
   ASC_duplicate_start,       // start can only be issued once
after create
```

```
    ASC_initialized_process,    // create does not want process
initialized with pid
    ASC_bad_path,               // path parm NULL, empty or too
long
    ASC_bad_remote_stdin_filename,  // remote_stdin_filename
has length 0 or is too long
    ASC_bad_remote_stdout_filename, // remote_stdout_filename
has length 0 or is too long
    ASC_bad_remote_stderr_filename, // remote_stderr_filename
has length 0 or is too long
    ASC_no_daemon_fd,           // daemon could not get an fd to
talk with created app
    ASC_bad_rem_infile_open,    // daemon could not open remote
stdin filename
    ASC_bad_rem_outfile_open,   // daemon could not open remote
stdout filename
    ASC_bad_rem_errfile_open,   // daemon could not open remote
stderr filename
    ASC_process_not_created,    // this process was not created
using Process::create
    ASC_process_not_attached,   // this process is not
currently attached
    ASC_remote_stdin_file,      // cannot send_stdin(), remote
stdin filename was specified
    ASC_no_destroy_from_connected,     // cannot issue destroy
when in connected state
    ASC_no_suspend_when_not_running,   // cannot suspend
process that is not running
    ASC_no_resume_when_running,       // cannot resume process
that is running
    ASC_no_sus_res_from_created,       // cannot suspend or
resume process from created state
    ASC_no_sus_res_from_connected,     // cannot suspend or
resume process from created state
```

```
    ASC_no_connect_from_created,       // cannot create
existing connected process
    ASC_no_disconnect_from_created,    // can issue start,
attach, destroy from created state
    ASC_no_detach_from_created,        // can issue start,
attach, destroy from created state
    ASC_no_detach_from_connected,      // can issue attach,
disconnect from connected state
    ASC_no_create_from_connected,      // cannot create process
from connected state
    ASC_no_create_from_attached,       // cannot create process
from attached state
    ASC_no_start_from_connected,       // cannot start process
from connected state
    ASC_no_start_from_attached,        // cannot start process
from attached state


//  PoeAppl related
    ASC_appl_has_no_procs,             // the application
contains no processies
    ASC_empty_att_cfg_file,            // found an empty attach
config file
    ASC_bad_att_cfg_version,           // error parsing version
in attach config file
    ASC_bad_att_cfg_numtask,           // error parsing number
of taks in attach config file
    ASC_bad_att_cfg_task,              // error parsing a task
number in attach config file
    ASC_bad_att_cfg_ipaddr,            // error parsing hte ip
address in the attach config file
    ASC_bad_att_cfg_hostname,          // error parsing the
hostname in attach config file
    ASC_bad_att_cfg_pid,               // error parsing the pid
in attach config file
```



```
    ASC_bad_att_cfg_sid,                // error parsing the sid
in the attach config file
    ASC_bad_att_cfg_progname,           // error parsing the
progname in the attach config file

//  module related
    ASC_module_not_found,       // module not found
    ASC_module_already_loaded,  // module already loaded
    ASC_module_already_unloaded,// module already unloaded
    ASC_module_invalid,         // module invalid
    ASC_not_expansible,         // source object not expansible
    ASC_expand_failed,          // source object expand failed

//  SD-Daemon
    ASC_daemon_communication_error,      // daemon
communication error
    ASC_daemon_create_error,    // daemon create error
    ASC_child_failed,           // child failed
    ASC_child_fork_failed,      // child fork failed
    ASC_exec_failed,            // exec failed
    ASC_failed_rhost_check,     // failed rhost check
    ASC_invalid_security_string, // invalid security string
    ASC_bad_userid,             // bad userid
    ASC_bad_groupid,            // bad groupid
    ASC_root_not_allowed,       // root not allowed
    ASC_identd_failed,          // identd failed
    ASC_security_init_failed,   // security initialization
failed
    ASC_security_failure,       // security failure
    ASC_no_access_allowed,      // no access allowed
    ASC_no_credentials,         // no credentials
```

```
        ASC_LAST_STATUS_VALUE
    };
```

Description

2.1.2 AisSeverityCode

Synopsis

```
#include <AisStatus.h>
enum AisSeverityCode {
     ASC_information,          //
     ASC_attention,            //
     ASC_error,                //
     ASC_severe,               //
     ASC_LAST_SEVERITY_VALUE
}
```

Description

2.2 Constructors

Synopsis

```
#include <AisStatus.h>
AisStatus(
     AisStatusCode status = ASC_success,
     AisSeverity severity = ASC_information)
AisStatus(const AisStatus &copy)
```

Parameters

`status`	Valid values are 0 code < `ASC_LAST_STATUS_VALUE`
`severity`	Valid values are 0 code <`ASC_LAST_SEVERITY_VALUE`

Description

Class constructor. This constructor initializes the object to reflect the specific status and severity codes.

Exceptions

An exception of type `AisStatus` with value `ASC_invalid_constructor` and severity `ASC_attention` is raised if the code is not a valid `AisStatusCode` value or the severity is not a valid `AisSeverityCode`.

2.3 add_data

Synopsis

```
#include <AisStatus.h>
void add_data(const char *data) const
```

Parameters

`data` a pointer to a character string representation of the data.

Description

This function adds one data value to the list of data associated with this condition.

See Also

`data_count, data_value, data_value_length`

2.4 data_count

Synopsis

```
#include <AisStatus.h>
int data_count(void) const
```

Description

This function returns the number of data values associated with this condition.

Return value

The count of data values reflected in the object.

2.5 data_value

Synopsis

```
#include <AisStatus.h>

char *data_value(int i, char *buffer, unsigned int len) const
```

Parameters

`i`	index value
`buffer`	caller-allocated buffer to hold the data value
`len`	maximum number of bytes the function will place in `buffer`. The `len` parameter should include enough space for a terminating *null* byte.

Description

A *null*-terminated string representation of the `i`th data value will be placed at the location specified by `buffer`. The value may be truncated if the `len` parameter is smaller than the length of the data value.

Return value

If the index is valid, that is, 0 `i` < `data_count`(), then a pointer to `buffer`, which will contain at most `len` bytes of the data value.

0 if the index is not valid.

See Also

`data_count, data_value_length`

2.6 data_value_length

Synopsis

```
#include <AisStatus.h>

unsigned int data_value_length(int i) const
```

Parameters

`i`	index value

Description

This function returns the length, including the terminating *null* byte, of the string representation of the `i`th data value.

Return value

If the index is valid, that is, 0 `i` < `data_count`(), then the length of the `i`th data value.

0 if the index is not valid.

See Also

`data_count, data_value`

2.7 operator =

Synopsis

```
#include <AisStatus.h>
AisStatus &operator = (const AisStatus &copy) const
```

Parameters

`copy`	object to be copied in the assignment

Description

This function copies the right hand side of the assignment expression over the left hand side.

Return value

A reference to the copied object, which is the left hand side of the assignment or the invoking object, depending upon the perspective.

2.8 operator AisStatusCode

Synopsis

```
#include <AisStatus.h>
operator AisStatusCode(void) const
```

Description

Cast function. This function returns the status code reflected in the object.

Return value

The status code in the object, of data type `AisStatusCode`.

2.9 operator int

Synopsis

```
#include <AisStatus.h>
operator int(void) const
```

Description

Cast function. This function returns the integer equivalent of the status code reflected in the object. A status value of zero reflects a “normal” status.

Return value

Integer equivalent of the status value `AisStatusCode`, and zero reflects “normal” status.

2.10 severity

Synopsis

```
#include <AisStatus.h>
AisSeverityCode severity(void) const
```

Description

Explicit severity function. This function returns the severity code reflected in the object.

Return value

The severity code in the object, of data type `AisSeverityCode`.

2.11 status

Synopsis

```
#include <AisStatus.h>
AisStatusCode status(void) const
```

Description

Explicit status function. This function returns the status code reflected in the object.

Return value

The status code in the object, of data type `AisStatusCode`.

2.12 status_name

Synopsis

```
#include <AisStatus.h>

const char *status_name(void) const
```

Description

This function returns the name of the status code reflected in the object. The name is in American English, and the string is stored in a constant array within the function. This function is intended only for limited diagnostic use during tool development.

Return value

The name of the status code in the object, of data type `char *`.

3.0 class Application

The Application class allows grouping a set of processies so that they can be acted upon together. This class contains a similar set of functions to the process class. Executing an application class function is generally the same as exuction the same function for each of the processies grouped within the Application class.

3.1 Constructors

Synopsis

```
#include <Application.h>
Application(void)
Application(const Application &copy)
```

Parameters

`copy`	object to be copied into the new `Application` object

Description

Default constructor.

The copy constructor uses the values contained in the `copy` argument to initialize the new (constructed) object.

Note: What functions in this base class should be virtual? All of them? None?

Exceptions

Exceptions that could be raised as a result of calling this function are unknown at this time.

`AisStatus`	???

3.2 activate_probe

Synopsis

```
#include <Application.h>
AisStatus activate_probe(
     short count,
     ProbeHandle *phandle,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`count`	number of probe expressions in the list to be activated
`phandle`	array of probe handles, one for each probe expression to be activated
`ack_cb_fp`	acknowledgement callback function to be invoked each time *all* probe expressions in the array have been activated (or activation fails) within a process
`ack_cb_tag`	tag to be used with the acknowledgement callback function

Description

This function activates a list of probes that have been installed within an application. The activation is atomic in the sense that all probes are activated or all probes fail to be activated for any given process within the application. Some processes within the application may successfully activate the probes while other processes fail, but within a process either all probes are successfully activated or none are activated. Probes are activated independently across processes, that is, there is no synchronization to ensure that the probes are activated in all processes at the same time.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

To activate a set of probes the processes must have been previously connected, and the probes must have been previously installed in those processes.

Note that `activate_probe` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until the probes have been activated or failed to be activated in all processes within the application. The acknowledgement callback function receives notification of the success or failure of the activation. The callback is activated once for each process within the application.

Return value

The return value indicates whether the requests for activation were successfully submitted, but indicates nothing about whether the requests themselves were successfully executed.

`ASC_success`	all activations were successfully submitted
`ASC_???`	

Callback Data

The callback function is invoked once for each process for which a probe activation is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	probes were successfully activated on this process
`ASC_operation_failed`	attempt to activate these probes in this process failed

See Also

`bactivate_probe`, `bconnect`, `bdisconnect`, `bprobe_deactivate`, `bprobe_install`, `class Process`, `connect`, `disconnect`, `GCBFuncType`, `probe_deactivate`, `probe_install`.

3.3 add_phase

Synopsis

```
#include <Application.h>
AisStatus add_phase(
     Phase ps,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)

AisStatus add_phase(
     Phase ps,
     ProbeExp init_func,
     GCBFuncType init_cb_fp,
     GCBTagType init_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`ps`	data structure local to the client containing the characteristics of the phase to be created
`init_func`	initialization function that is executed once within the application when the phase is installed
`init_cb_fp`	callback function to handle messages from the initialization function
`init_cb_tag`	tag to be used with the initialization callback function
`ack_cb_fp`	acknowledgement callback function to be invoked each time the phase has been created within a process
`ack_cb_tag`	tag to be used with the acknowledgement callback function

Description

This function adds a new phase structure to each connected process within the application. A process *must* be connected in order to add a new phase. The phase does not execute for the first time until the amount of time indicated by the phase period has elapsed, starting from the time the phase is added to the process.

Note that `add_phase` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until the phase has been installed or failed to be installed in all processes within the application. The acknowledgement callback function receives notification of the success or failure of the installation. The callback is activated once for each process within the application.

The initialization function must be loaded into the application before this operation may take place. The function prototype for the initialization function is:

- `void init_func(void *msg_handle)`

Return value

The return value indicates whether the requests for phase addition were successfully submitted, but indicates nothing about whether the requests themselves were successfully executed.

`ASC_success`	all phase additions were successfully submitted
`ASC_operation_failed`	attempt to add a phase to some process failed, perhaps because the process is not connected

Callback Data

`init_cb_fp`. This callback function is invoked each time the corresponding function in the process instrumentation -- `init_func` -- sends a message to the client. The message format is determined by the function that sends the message.

`ack_cb_fp`. The callback function is invoked once for each process for which a phase addition is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	phase was successfully added to this process
`ASC_operation_failed`	attempt to add a phase to this process failed, perhaps because the phase is already added to the process

See Also

`badd_phase, bconnect, bdisconnect, class GenCallBack, class ProbeMod, class Process, connect, disconnect, GCBFuncType, GCBTagType, Process::alloc_mem, Process::free_mem.`

3.4 add_process

Synopsis

```
#include <Application.h>

AisStatus add_process(const Process *p)
```

Parameters

`p`	process to be added to the application

Description

This function adds a process to the set of processes managed by the application. This operation acts locally within the end-user tool. It does not attempt to connect to the process. The process state (e.g. connected or attached) is not required to match the state of all other processes within the application.

The index of a process is not guaranteed to remain invariant when new processes are added to or removed from an application. The index does remain invariant otherwise.

Return value

The return value indicates whether the process addition was successful.

`ASC_success`	process was successfully added
`ASC_operation_failed`	attempt to add this process to this application failed

See Also

`connect, bconnect, bdisconnect, disconnect, remove_process.`

3.5 alloc_mem

Synopsis

```
#include <Application.h>
ProbeExp alloc_mem(
     ProbeType pt,
     void *init_val,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag,
     AisStatus &stat)

ProbeExp alloc_mem(
     ProbeType pt,
     void *init_val,
     Phase ps,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag,
     AisStatus &stat)
```

Parameters

`pt`	data type of the allocated data
`init_val`	pointer to the initial value of the allocated data, or 0 if no initial value is desired
`ps`	phase that will contain the allocated data
`ack_cb_fp`	callback function to process acknowledgement messages
`ack_cb_tag`	tag to be used as an argument to the acknowledgement callback when it is invoked
`stat`	output value indicating the completion status of the function

Description

This function allocates a block of probe data in each process in the application. It returns a single probe expression that may be used to reference the allocated data. The data may be referenced in a probe expression that may be installed in any or all of the application processes where the data is allocated.

Note that alloc_mem returns control to the caller immediately and does not wait until it has either succeeded or failed on all of the processes within the application. The probe expression representing the allocation is returned immediately whether or not the allocations succeed. The returned probe expression may be used as a data reference on any process where the allocation succeeds. If the data reference is used in another probe expression and the client attempts to install that probe expression in a process where the allocation failed, that probe expression will fail to install. Similarly, installation will fail if one attempts to install the probe in a process where the data was not allocated.

Stat indicates whether all requests for allocation were successfully submitted. If all requests are successfully submitted stat is given the value ASC_success. If some request cannot be submitted then stat is given the value ASC_operation_failed. It reflects the highest severity encountered.

Return value

A probe expression that may be used as a valid reference to the data on any process in which the data has been successfully allocated.

Callback Data

The callback function is invoked once for each process for which data allocation is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type AisStatus, which contains one of the following status values:

ASC_success	data was successfully allocated in this process
ASC_operation_failed	attempt to allocate data in this process failed

See Also

bfree_mem, balloc_mem, free_mem, status

3.6 attach

Synopsis

```
#include <Application.h>
AisStatus attach(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to be invoked with each successful or failed attachment to a process listed within the application.
`tag`	callback tag to be used as a parameter to the callback each time the callback function is invoked.

Description

Attach to all processes within an application. When multiple tools are connected to a process or application, only one tool can be attached at a time. Attaching to a process or application allows the tool to control the execution directly such as, suspending and resuming execution. Processes must first be connected or created before they can be attached.

Note that `attach` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until all processes within the application have attached or failed to attach. The acknowledgement callback function receives notification of the success or failure of the activation. The callback is activated once for each process within the application.

Return value

The return value for `attach` indicates whether the requests were successfully submitted, but indicates nothing about whether the requests themselves were successfully executed.

`ASC_success`	all requests to attach were successfully submitted
`ASC_operation_failed`	attempt to request attachment to some process failed, perhaps because the process is not connected

Callback Data

The callback function is invoked once for each process for which an attach is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully attached
`ASC_operation_failed`	attempt to attach to this process failed
ASC_duplicate_attach	already attached

See Also

`battach, bdetach, detach`

3.7 bactivate_probe

Synopsis

```
#include <Application.h>
AisStatus bactivate_probe(short count, ProbeHandle *phandle)
```

Parameters

`count`	number of probe expressions in the list to be activated
`phandle`	array of probe handles, one for each probe expression to be activated

Description

This function activates a list of probes that have been installed within an application. The activation is atomic in the sense that all probes are activated or all probes fail to be activated for any given process within the application. Some processes within the application may successfully activate the probes while other processes fail, but within a process either all probes are successfully activated or none are activated. Probes are activated independently across processes, that is, there is no synchronization to ensure that the probes are activated in all processes at the same time.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

To activate a set of probes the processes must have been previously connected, and the probes must have been previously installed in those processes.

Note that the function submits the requests to activate the probes and waits until the requests have completed. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value indicates whether *all* of the requests for activation were successfully executed. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all activations were successfully completed
`ASC_operation_failed`	one or more of the activations failed

See Also

`activate_probe, bconnect, bdisconnect, bprobe_deactivate, bprobe_install, connect, disconnect, probe_deactivate, probe_install.`

3.8 badd_phase

Synopsis

```
#include <Application.h>
AisStatus badd_phase(Phase ps)
AisStatus badd_phase(
     Phase ps,
     ProbeExp init_func,
     GCBFuncType init_cb_fp,
     GCBTagType init_cb_tag)
```

Parameters

`ps`	data structure local to the client containing the characteristics of the phase to be created
`init_func`	initialization function that is executed once within the application when the phase is installed
`init_cb_fp`	callback function to handle messages from the initialization function
`init_cb_tag`	tag to be used with the initialization callback function

Description

This function adds a new phase structure to each connected process within the application. A process *must* be connected in order to add a new phase. The phase does not execute for the first time until the amount of time indicated by the phase period has elapsed, starting from the time the phase is added to the process.

Note that the function submits the requests to add the phase and waits until the requests have completed. The return value indicates whether *all* of the requests were successfully executed. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

The initialization function must be loaded into the application before this operation may take place. The function prototype for the initialization function is:

- `void init_func(void *msg_handle)`

Return value

The return value indicates whether requests to all processes for phase addition were successfully executed. The return value reflects the highest severity encountered across all processes.

`ASC_success`	phase was successfully added to all processes
`ASC_operation_failed`	one or more of the phase additions failed

Callback Data

The callback function is invoked each time the corresponding function in the process instrumentation -- `init_func` -- sends a message to the client. The message format is determined by the function that sends the message.

See Also

`add_phase, bconnect, bdisconnect, class ProbeMod, connect, disconnect, Process::alloc_mem, Process::free_mem.`

3.9 balloc_mem

Synopsis

```
#include <Application.h>

ProbeExp balloc_mem(ProbeType pt, void *init_val, AisStatus
&stat)


ProbeExp balloc_mem(
    ProbeType pt,
    void *init_val,
    Phase ps,
    AisStatus &stat)
```

Parameters

`pt`	data type of the allocated data
`init_val`	pointer to the initial value of the allocated data, or 0 if no initial value is desired
`ps`	phase that will contain the allocated data
`stat`	output value indicating the completion status of the function

Description

This function allocates a block of probe data in each process in the application. It returns a single probe expression that may be used to reference the allocated data. The data may be referenced in a probe expression that may be installed in any or all of the application processes where the data is allocated. The initial value of the data is as specified, or zero if not specified.

Note that `balloc_mem` does not return control to the caller until it has either succeeded or failed on all of the processes within the application. If the allocation succeeds it returns a valid probe expression data reference and `stat` is given the value `ASC_success`. If the allocation fails on some process then `stat` is given the value `ASC_operation_failed` and any probe that references the returned value of `balloc_mem` will fail to install on that process.

The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

A probe expression that may be used as a valid reference to the data on any process in which the data has been successfully allocated.

See Also

`bfree_mem, free_mem, alloc_mem, status`

3.10 battach

Synopsis

```
#include <Application.h>
AisStatus battach(void)
```

Description

Attach to all processes within an application. When multiple tools are connected to a process or application, only one tool can be attached at a time. Attaching to a process or application allows the tool to control the execution directly, setting break points, starting, suspending and resuming execution, *etc*. Processes must first be connected or created before they can be attached.

Note that `battach` does not return control to the caller until all attachments have either succeeded or failed. The return value indicates whether all succeeded or some succeeded and some failed. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `battach` indicates whether the individual attachments themselves were successfully established. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all processes were successfully attached as expected.
`ASC_operation_failed`	one or more of the processes failed to attach
ASC_duplicate_attach	already attached

See Also

`attach, bdetach, detach`

3.11 bconnect

Synopsis

```
#include <Application.h>
AisStatus bconnect(void)
```

Description

Connect to all processes within an application. Connection to a process establishes a communication channel to the CPU where the process resides and creates the environment within that process that allows the client to insert and remove instrumentation, alter its control flow, *etc*.

Connections from multiple DPCL based tools to the same processies within the application are allowed.

Note that `bconnect` does not return control to the caller until all connections have either succeeded or failed. The return value indicates whether all connections succeeded or some succeeded and some failed. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bconnect` indicates whether the connections themselves were successfully established. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all connections were successfully established as expected.
`ASC_operation_failed`	one or more of the connections failed to be established.

See Also

`bdisconnect, connect, disconnect, PoeAppl::binit_procs, PoeAppl::init_procs`

3.12 bdeactivate_probe

Synopsis

```
#include <Application.h>
AisStatus bdeactivate_probe(short count, ProbeHandle *phandle)
```

Parameters

`count`	number of probes to be deactivated
`phandle`	array of probe handles, representing the probes, to be deactivated

Description

This function accepts an array of probe handles as an input parameter. Each probe handle in the array represents a probe that has been installed in the application. The client sends a request to each of the processes within the application to deactivate the list of probes represented by the array. Probes are deactivated atomically for each process in the sense that the process is temporarily stopped, all probes on the list are deactivated, then the process is restarted. None of the probes in the array are left active.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

Note that `bdeactivate_probe` does not return control to the caller until all probes in the array have been deactivated on all processes in the application. The return value indicates whether all connections succeeded or some succeeded and some failed. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bdeactivate_probe` indicates whether the deactivations were successfully completed. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all probe deactivations completed as expected
`ASC_operation_failed`	one or more of the probe deactivations failed

See Also

3.13 bdestroy

Synopsis

```
#include <Application.h>
AisStatus bdestroy(void)
```

Description

This function destroys or terminates all processes within the application.

If this is called from a PoeAppl object, the poe process itself is also destroyed.

Note that `bdestroy` does not return control to the caller until all processes within the application have been destroyed. The return value indicates whether all terminations succeeded or some succeeded and some failed. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bdestroy` indicates whether the terminations were successfully completed. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all terminations were successfully completed, as expected
ASC_no_destroy_from_connected	process must be in attached state to call destory
`ASC_operation_failed`	one or more of the terminations failed

See Also

`destroy`

3.14 bdetach

Synopsis

```
#include <Application.h>
AisStatus bdetach(void)
```

Description

This function detaches all processes in the application. Process control flow, such suspending and resuming processies, can only be done while a process is in an attached state. Detaching a process removes the level of process control available to the client or tool when the process is attached, but retains the process connection so probe installation, activation, removal, *etc*. can still take place.

Note that `bdetach` does not return control to the caller until all processes within the application have been detached. The return value indicates whether all processes successfully detached or some succeeded and some failed. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bdetach` indicates whether all processes were successfully detached. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all processes were successfully detached, as expected
ASC_no_detach_from_created	currently created, must attach before detaching
ASC_no_detach_from_connected	currently connected, must attach before detaching
`ASC_operation_failed`	one or more processes failed to detach

See Also

`attach, battach, detach`

3.15 bdisconnect

Synopsis

```
#include <Application.h>
AisStatus bdisconnect(void)
```

Description

Disconnect from all processes within an application. Disconnecting from an application process removes the application environment created by a connection. All instrumentation and data are removed from the application process.

Note that `bdisconnect` does not return control to the caller until all processes within the application have either succeeded or failed in disconnecting. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bdisconnect` indicates whether the connections were successfully terminated. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all connections were successfully terminated as expected
`ASC_operation_failed`	one or more of the connections failed to terminate

See Also

`disconnect, connect, bconnect`

3.16 bexecute

Synopsis

```
#include <Application.h>
AisStatus bexecute(
     ProbeExp pexp,
     GCBFuncType data_cb_fp,
     GCBTagType data_cb_tag)
```

Parameters

`pexp`	probe expression to be executed in the application process
`data_cb_fp`	callback function to be invoked when data from the probe is received
`data_cb_tag`	callback tag to be used when the data callback function is invoked

Description

This function executes a probe expression in each process within an application. The expression is executed once in each process, then removed. The application process is interrupted, the expression is executed, then the process resumes execution as before the interruption.

Note that `bexecute` does not return control to the caller until the probe expression has either succeeded or failed to execute within all processes in an application. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bexecute` indicates whether the execution succeeded or failed.

`ASC_success`	probe expression was successfully executed
`ASC_operation_failed`	attempt to execute the probe expression failed

See Also

`execute`

3.17 bfree_mem

Synopsis

```
#include <Application.h>

AisStatus bfree_mem(ProbeExp pexp)
```

Parameters

`pexp`	dynamically allocated block of probe memory

Description

This function deallocates a block of dynamically allocated probe memory for every process in the application. The probe expression must contain only a single reference to a block of data allocated by the `alloc_mem` or `balloc_mem` functions.

Note that `bfree_mem` does not return control to the caller until all processes within the application have either succeeded or failed in deallocating the block of memory. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bfree_mem` indicates whether all requests for deallocation were successfully executed. The return value reflects the highest severity encountered across all processes.

See Also

`free_mem, balloc_mem, alloc_mem`

3.18 binstall_probe

Synopsis

```
#include <Application.h>
AisStatus binstall_probe(
     short count,
     ProbeExp *probe_exp,
     InstPoint *point,
     GCBFuncType *data_cb_fp,
     GCBTagType *data_cb_tag,
     ProbeHandle *phandle)
```

Parameters

`count`	number of probe expressions to be installed
`probe_exp`	probe expressions to be installed
`point`	instrumentation points where the probe expressions are to be installed
`data_cb_fp`	callback functions to process data received from the probe expression
`data_cb_tag`	tags to be used as an argument to the data callback when it is invoked
`phandle`	probe handles that represent the installed probe expressions

Description

This function installs probe expressions as instrumentation at specific locations within each process in the application. Probe expressions are installed atomically, in the sense that within each process either all probe expressions in the request are installed into the process, or none of the expressions are installed. There is no synchronization across processes to assure that all processes install all probes. The return value indicates whether all probes were installed, or whether one or more processes were unable to install the expressions as requested.

`Data_cb_fp` is an input array supplied by the caller that must contain at least `count` elements. The i^{th} element of the array is a pointer to a callback function that is invoked each time the i^{th} probe in `phandle` sends data via the `AisSendMsg` function. `Data_cb_tag` is a similar array that contains the callback tag used when callbacks in `data_cb_fp` are invoked. The i^{th} callback tag is used with the i^{th} callback.

`Phandle` is an output array supplied by the caller that must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, to be used in subsequent references to the i^{th} probe expression. For example, it is needed when the client activates, deactivates or removes a probe expression from an application or process. `Phandle` does not contain valid information if the installation fails.

Note that `binstall_probe` does not return control to the caller until all probe expressions have been installed or failed to install within all processes within the application.The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `binstall_probe` indicates whether the probe installations were successful. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all probes were successfully installed, as expected
`ASC_operation_failed`	one or more of the probes could not be installed as requested, so none of the probes were installed

Callback Data

The callback function is invoked once for each message sent from the probe. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback tag is given in the `data_cb_tag` array. The callback message is the data send by the probe using the `Ais_send()` function call.

See Also

`AisSendMsg, install_probe, ...`

3.19 bload_module

Synopsis

```
#include <Application.h>
AisStatus bload_module(ProbeModule* module)
```

Parameters

module	the probe module to be loaded.

Description

This function sends and loads the module from the client side to all the processes within the Application class. Once loaded, the probe expressions available in this probe module can be installed and activated as if those are native in the application.

Note that `bload_module` does not return control to the caller until the probe module has been installed or failed to install in all processes within the application.The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bload_module` indicates whether the probe module installations were successful. The return value reflects the highest severity encountered across all processes.

`ASC_success`	module was successfully installed on all processes
`ASC_operation_failed`	module could not be installed as requested on one or more processes

See Also

`bunload_module, load_module, unload_module`

3.20 bremove_phase

Synopsis

```
#include <Application.h>
AisStatus bremove_phase(Phase ps)
```

Parameters

`ps`	phase description to be removed from the application

Description

This function removes a phase from the application. Data and functions associated with the phase are unaffected by removing the phase.

Note that `bremove_phase` does not return control to the caller until the phase has been removed or failed to be removed from all processes within the application.The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bremove_phase` indicates whether the phase was successfully removed from all processes. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all phases were successfully removed, as expected
`ASC_operation_failed`	phase could not be removed from one or more processes

See Also

`add_phase, badd_phase, class Phase, remove_phase`

3.21 bremove_probe

Synopsis

```
#include <Application.h>
AisStatus bremove_probe(short count, ProbeHandle *phandle)
```

Parameters

`count`	number of probe handles in the accompanying array
`phandle`	array of probe handles representing probe expressions to be removed

Description

This function deletes or removes probe expressions that have been installed in an application. If all probe expressions are installed and deactivated, the probe expressions are removed and a "normal" return status results. If one or more of the probe expressions are currently active, the expressions are deactivated and removed, and the return status indicates there were active probes at the time of their removal. If one or more of the probes do not exist, all existing probes are removed and the return status indicates an appropriate warning. If one or more of the probe expressions exists but cannot be removed, an error results and as many probes as can be are removed. If one or more processes are not connected, probe removal takes place within those that are connected, and a warning is issued.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

Probe expression removal is atomic in the sense that all probe expressions are removed from a given process or none are. When probes are removed from a process the process is temporarily stopped, all indicated probes are removed, and the process is resumed. Probe expressions are removed in a process by process basis. There is no synchronization between processes to guarantee that all expressions are removed from all processes. One process may succeed while another one fails.

Note that `bremove_probe` does not return control to the caller until the probes have been removed or failed to be removed from all processes within the application.The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bremove_probe` indicates whether all probes in the list were successfully removed from all processes. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all probes were successfully removed, as expected
`ASC_operation_failed`	none of the probes were removed

See Also

`bactivate_probe`, `bdeactivate_probe`, `binstall_probe`, `activate_probe`, `deactivate_probe`, `install_probe`, `remove_probe`

3.22 bresume

Synopsis

```
#include <Application.h>
AisStatus bresume(void)
```

Description

This function resumes execution of an application that has been temporarily suspended by a `suspend` or `bsuspend` function. Execution resumption occurs on a process by process basis. A process must be attached for it to be resumed. A resume issued against a process that is not attached will result in a warning return code.

Note that `bresume` does not return control to the caller until the all processes within the application have resumed or failed to resume.The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bresume` indicates whether all processes were successfully resumed. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all processes were resumed, as expected
`ASC_operation_failed`	some processes failed to be resumed
ASC_no_sus_res_from_created	must be attached to call bresume
ASC_no_sus_res_from_connected	must be attached to call bresume

See Also

`attach, battach, bconnect, bdetach, bdisconnect, bsuspend, connect, detach, disconnect, resume, suspend`

3.23 bset_phase_exit

Synopsis

```
#include <Application.h>
AisStatus bset_phase_exit(
     Phase ps,
     ProbeExp begin_func,
     GCBFuncType begin_cb_fp,
     GCBTagType begin_cb_tag,
     ProbeExp iter_func,
     GCBFuncType iter_cb_fp,
     GCBTagType iter_cb_tag,
     ProbeExp end_func,
     GCBFuncType end_cb_fp,
     GCBTagType end_cb_tag)
```

Parameters

`ps`	phase description to be removed from the application
`begin_func`	initialization function that is executed once within the application when the phase is removed
`begin_cb_fp`	callback function to handle messages from the initialization function
`begin_cb_tag`	tag to be used with the initialization callback function
`iter_func`	iteration function that is executed within the application on each piece of data associated with the phase when the phase is removed
`iter_cb_fp`	callback function to handle messages from the iteration function
`iter_cb_tag`	tag to be used with the iteration callback function
`end_func`	termination function that is executed once within the application when the phase is removed
`end_cb_fp`	callback function to handle messages from the termination function
`end_cb_tag`	tag to be used with the termination callback function

Description

This function specifies a set of exit functions to be executed when any of the following three events occur.

- when the indicated phase is removed using either the remove_phase or bremove_phase function call
- when diconnecting from the target application (without calling remove_phase or bremove_phase first)
- when the target application has finished execution while the indicated phase is still active

Note that `set_phase_exit` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the exit functions have been placed in the indicated phase or the operation failed to complete.

Each of the phase functions must be loaded into the application before this operation may take place. The function prototypes for the functions are:

- `void begin_func(void *msg_handle)`
- `void iter_func(void *msg_handle, void *data)`
- `void end_func(void *msg_handle)`

Return value

The return value for `remove_phase` indicates whether the requests to remove the indicated phase on all processes in the application were successfully submitted. It gives no indication of whether the requests were successfully executed.

`ASC_success`	all remove requests were successfully submitted
`ASC_operation_failed`	remove operation failed to be requested to some process

Callback Data

begin_cb_fp, **iter_cb_fp**, **end_cb_fp**. These callback functions are invoked each time the corresponding function in the process instrumentation -- `begin_func`, `iter_func`, or `end_func` -- sends a message to the client. The message format is determined by the function that sends the message.

See Also

`set_phase_exit, add_phase, badd_phase, remove_phase, bremove_phase`

3.24 bset_phase_period

Synopsis

```
#include <Application.h>

AisStatus bset_phase_period(Phase ps, float period)
```

Parameters

`ps`	phase to be modified
`period`	new time interval between successive phase activations, in seconds

Description

This function changes the time interval between successive activations of a phase. The interval change occurs on a process by process basis for all processes within the application. Processes which do not have the phase installed result in an informational return code. Processes that are not connected result in a warning return code.

The new period is represented by a floating-point value. If the value is positive it represents the time interval in seconds. If the value is zero or positive and smaller than the minimum activation time interval, it represents the minimum activation delay time. In both cases the phase is activated immediately before setting the new interval. If the value is less than zero the phase is disabled immediately, but left in place for possible future reactivation.

Note that `bset_phase_period` does not return control to the caller until the phase period has been set or failed to be set in all processes within the application.The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bset_phase_period` indicates whether the phase period was successfully set on all processes. The return value reflects the highest severity encountered across all processes.

`ASC_success`	phase period was successfully set on all processes
`ASC_operation_failed`	some processes failed to set the phase period

See Also

`add_phase, badd_phase, bremove_phase, get_phase_period, remove_phase, set_phase_period`

3.25 bsignal - LY

Synopsis

```
#include <Application.h>
AisStatus bsignal(int unix_signal)
```

Parameters

`unix_signal`	Unix™ signal to be sent to every process in the application

Description

This function sends the specified signal to every process in the application. The process must be both connected and attached to receive the signal. The function does not return until all processes in the application have received the signal.

A signal is sent only to those processes that are connected and attached.

Note that `bsignal` does not return control to the caller until each process within the application has been signalled or failed to be signalled.The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bsignal` indicates whether the AIX signal was successfully sent to all processes. The return value reflects the highest severity encountered across all processes.

`ASC_success`	signal was successfully sent to all processes
`ASC_operation_failed`	signal failed to be sent to one or more processes

See Also

`signal`

3.26 bstart

Synopsis

```
#include <Application.h>
AisStatus bstart(void)
```

Description

This function starts the execution of an application that has been created but not yet begun execution. It does this by issueing a start to each process contained in the application.

To get a created application running, it is required to issue one start. Subsequent starts cannot be issued.

Note that `bstart` does not return control to the caller until the application has started or failed to start. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bstart` indicates whether the application was successfully started.

`ASC_success`	application was started
`ASC_operation_failed`	application failed to be started
ASC_destroyed_process	a process has been destroyed
ASC_disconnecting_process	a process is disconnecting
ASC_duplicate_start	start can only be issued once after create

See Also

`Process::bcreate, bdestroy, Process::create, destroy, start, PoeAppl class`

3.27 bsuspend

Synopsis

```
#include <Application.h>
AisStatus bsuspend(void)
```

Description

This function suspends an application that is executing. Application suspension occurs on a process by process basis. A tool must be attached to a process in order to suspend process execution.

Note that `bsuspend` does not return control to the caller until each process within the application has been suspended or failed to be suspended.The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bsuspend` indicates whether all processes within the application were successfully suspended. The return value reflects the highest severity encountered across all processes.

`ASC_success`	all processes were successfully suspended
`ASC_operation_failed`	one or more processes failed to be suspended
ASC_no_sus_res_from_created	must be attached to call bsuspend
ASC_no_sus_res_from_connected	must be attached to call bsuspend

See Also

`bresume, resume, suspend`

3.28 bunload_module

Synopsis

```
#include <Application.h>
AisStatus bunload_module(ProbeModule *module)
```

Parameters

`module`	probe module to be removed from each application process

Description

This function unloads the module from all the processes within the Application class. Once unloaded, All the probe handles that refer to this probe module are automatically removed.

Note that `bunload_module` does not return control to the caller until the probe module has been removed or failed to be removed from all processes within the application.The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

Return value

The return value for `bunload_module` indicates whether the probe module was successfully removed from all processes. The return value reflects the highest severity encountered across all processes.

`ASC_success`	module was successfully removed from all processes
`ASC_operation_failed`	module could not be removed from one or more processes

See Also

`bload_module, load_module, unload_module`

3.29 connect

Synopsis

```
#include <Application.h>
AisStatus connect(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to be invoked with each successful or failed connection to a process listed within the application
`tag`	callback tag to be used each time the callback function is invoked

Description

Connect to all processes within an application. Connection to a process establishes a communication channel to the machine where the process resides and creates the environment within that process that allows the client to insert and remove instrumentation, alter its control flow, *etc*.

Connections from multiple DPCL based tools to the same processies within the application are allowed.

Note that the function submits the requests to connect the processes and returns immediately. The callback function receives notification of each connection's success or failure.

Return value

The return value for `connect` indicates whether the requests for connection were successfully submitted, but indicates nothing about whether the requests themselves were successfully executed.

`ASC_success`	request for connection was successfully sent
`ASC_operation_failed`	attempt to send request to connect to this process failed

Callback Data

The callback function is invoked once for each process for which a connection is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	connection was successfully established on this process
`ASC_operation_failed`	attempt to connect to this process failed

See Also

`bconnect, bdisconnect, disconnect, PoeAppl::init_procs, PoeAppl::binit_procs`

3.30 deactivate_probe

Synopsis

```
#include <Application.h>
AisStatus deactivate_probe(
     short count,
     ProbeHandle *phandle,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`count`	number of probes to be deactivated
`phandle`	array of probe handles, representing the probes, to be deactivated
`ack_cb_fp`	acknowledgement callback function to be invoked each time *all* probe expressions in the array have been deactivated (or deactivation fails) within a process
`ack_cb_tag`	tag to be used with the acknowledgement callback function

Description

This function accepts an array of probe handles as an input parameter. Each probe handle in the array represents a probe that has been installed in the application. The client sends a request to each of the processes within the application to deactivate the list of probes represented by the array. Probes are deactivated atomically for each process in the sense that the process is temporarily suspended, all probes on the list are deactivated, then the process is restarted. None of the probes in the array are left active.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

Note that `deactivate_probe` returns control immediately to the caller. It does not wait until all probes in the array have been deactivated on all processes in the application. The return value indicates whether all requests were successfully submitted and gives no indication whatever about the success or failure of the execution of those requests.

Return value

The return value for `deactivate_probe` indicates whether the deactivations were successfully submitted.

`ASC_success`	all probe deactivations were submitted, as expected
`ASC_operation_failed`	one or more of the probe deactivations were not submitted

Callback Data

The callback function is invoked once for each process for which a probe deactivation is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	probes were successfully deactivated on this process
`ASC_operation_failed`	attempt to deactivate probes on this process failed

See Also

3.31 destroy

Synopsis

```
#include <Application.h>
AisStatus destroy(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	acknowledgement callback function to be invoked for each process that is destroyed (or not destroyed)
`tag`	tag to be used with the acknowledgement callback function

Description

This function destroys or terminates all processes within the application.

If this is called from a PoeAppl object, the poe process itself is also destroyed.

Note that `destroy` returns control to the caller immediately. It does not wait until all processes within the application have been destroyed. The return value indicates whether the requests were successfully submitted, but gives no indication of whether the requests themselves were successfully executed.

Return value

The return value for `destroy` indicates whether the terminations were successfully requested.

`ASC_success`	all terminations were successfully requested, as expected
`ASC_operation_failed`	one or more of the terminations were not requested

Callback Data

The callback function is invoked once for each process for which destruction is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully destroyed
ASC_no_destroy_from_connected	process must be in attached state to call destory
`ASC_operation_failed`	attempt to destroy this process failed

See Also

`bdestroy`

3.32 detach

Synopsis

```
#include <Application.h>
AisStatus detach(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to be invoked with each successful or failed detachment from a process listed within the application.
`tag`	callback tag to be used each time the callback function is invoked.

Description

This function detaches all processes in the application. Process control flow, such as suspending and resuming processies, can only be done while a process is in an attached state. Detaching a process removes the level of process control available to the client or tool when the process is attached, but retains the process connection so probe installation, activation, removal, *etc*. can still take place.

Note that `detach` returns control to the caller immediately upon issuing all requests to detach from the processes. The return value indicates whether all requests were successfully submitted.

Return value

The return value for `detach` indicates whether all requests were successfully submitted.

`ASC_success`	all detach requests were successfully submitted, as expected
`ASC_operation_failed`	one or more requests were not submitted

Callback Data

The callback function is invoked once for each process for which detachment is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully detached
ASC_no_detach_from_created	currently created, must attach before detaching
ASC_no_detach_from_connected	currently connected, must attach before detaching
`ASC_operation_failed`	attempt to detach this process failed

See Also

`attach, battach, bdetach`

3.33 disconnect

Synopsis

```
#include <Application.h>
AisStatus disconnect(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to be invoked with each successful or failed disconnection from a process listed within the application.
`tag`	callback tag to be used each time the callback function is invoked.

Description

Disconnect from all processes within an application. Disconnecting from an application process removes the application environment created by a connection. All instrumentation and data are removed from the application process.

Note that the function submits the requests to disconnect the processes and returns immediately. The callback function receives notification of each disconnection's success or failure.

Return value

The return value for `disconnect` indicates whether the requests for disconnection were successfully submitted, but indicates nothing about whether the requests themselves were successfully executed.

Callback Data

The callback function is invoked once for each process for which disconnection is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully disconnected
`ASC_operation_failed`	attempt to disconnect this process failed

See Also

`bconnect, bdisconnect, connect`

3.34 execute

Synopsis

```
#include <Application.h>
AisStatus execute(
     ProbeExp pexp,
     GCBFuncType data_cb_fp,
     GCBTagType data_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`pexp`	probe expression to be executed in the application process
`data_cb_fp`	callback function to be invoked when data from the probe is received
`data_cb_tag`	callback tag to be used when the data callback function is invoked
`ack_cb_fp`	callback function to be invoked when execution succeeds or fails
`ack_cb_tag`	callback tag to be used when the callback function is invoked

Description

This function executes a probe expression within all application processes within an application. The expression is executed once, then removed. The application process is interrupted, the expression is executed, then the process resumes execution as before the interruption.

Note that `execute` returns control to the caller immediately upon submitting its request to the daemons. It does not wait until the probe expression has been executed or failed to execute. The acknowledgement callback function receives notification of the success or failure of the execution. The callback is executed once for each process within the application.

Return value

The return value for `execute` indicates whether the request for deallocation was successfully submitted, but indicates nothing about whether the request was successfully executed.

`ASC_success`	probe expression execution was successfully submitted
`ASC_???`	

Callback Data

The callback function is invoked when execution succeeds or fails. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	probe expression was successfully executed
`ASC_operation_failed`	attempt to execute the probe expression failed

See Also

`bexecute`

3.35 free_mem

Synopsis

```
#include <Application.h>
AisStatus free_mem(
    ProbeExp pexp,
    GCBFuncType ack_cb_fp,
    GCBTagType ack_cb_tag)
```

Parameters

`pexp`	dynamically allocated block of probe memory
`ack_cb_fp`	callback function to be invoked when deallocating the block of memory succeeds or fails
`ack_cb_tag`	callback tag to be used when the callback function is invoked

Description

This function deallocates a block of dynamically allocated probe memory for every process in the application. The probe expression must contain only a single reference to a block of data allocated by the `alloc_mem` or `balloc_mem` functions.

Note that `free_mem` returns control to the caller immediately upon submitting its request to free the data. It does not wait until the data has been deallocated or failed to deallocate. The acknowledgement callback function receives notification of the success or failure of the deallocation. The callback is executed once for each process within the application.

Return value

The return value for `free_mem` indicates whether the requests for deallocation were successfully submitted, but indicates nothing about whether the requests themselves were successfully executed.

Callback Data

The callback function is invoked once for each process for which deallocation is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	block of probe memory was successfully deallocated
`ASC_operation_failed`	attempt to deallocate memory on this process failed

See Also

`bfree_mem, balloc_mem, alloc_mem`

3.36 get_count

Synopsis

```
#include <Application.h>
int get_count(void) const
```

Description

This function returns the number of processes currently included in the application.

Return value

The number of `Process` objects in the application.

See Also

`get_process, status`

3.37 get_process

Synopsis

```
#include <Application.h>

Process get_process(int i) const
```

Parameters

`i`	the position or index into the process table whose entry is to be retrieved.

Description

Returns the i^{th} Process object of the application.

Return value

The i^{th} Process object if the index is valid, that is, 0 `i` < `get_count`() or an invalid process if the index is not valid.

See Also

`get_count`

3.38 install_probe

Synopsis

```
#include <Application.h>
AisStatus install_probe(
     short count,
     ProbeExp *probe_exp,
     InstPoint *point,
     GCBFuncType *data_cb_fp,
     GCBTagType *data_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag,
     ProbeHandle *phandle)
```

Parameters

`count`	number of probe expressions to be installed, instrumentation points, data callback functions, data callback tags, and probe handles
`probe_exp`	probe expressions to be installed
`point`	instrumentation points where the probe expressions are to be installed
`data_cb_fp`	callback function to process data received from the probe expression
`data_cb_tag`	tag to be used as an argument to the data callback when it is invoked
`ack_cb_fp`	callback function to process installation acknowledgments
`ack_cb_tag`	tag to be used as an argument to the acknowledgement callback when it is invoked
`phandle`	probe handles that represent the installed probe expressions

Description

This function installs probe expressions as instrumentation at specific locations within each process in the application. Probe expressions are installed atomically, in the sense that within each process either all probe expressions in the request are installed into the process, or none of the expressions are installed. There is no synchronization across processes to assure that all processes install all probes. The return value indicates whether all requests to have probes installed were successfully submitted.

`Phandle` is an output array supplied by the caller that must contain at least `count` elements. The ith element of the array is a handle, or identifier, to be used in subsequent references to the ith probe expression. For example, it is needed when the client activates, deactivates or removes a probe expression from an application or process. `Phandle` does not contain valid information if the installation fails.

Note that `install_probe` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until all probe expressions have been installed or failed to install within all processes within the application.

Return value

The return value for `install_probe` indicates whether the requests for probes to be installed were successfully submitted. It gives no indication of whether those requests were successfully executed.

`ASC_success`	all probe expression installation requests were successfully submitted
`ASC_operation_failed`	one or more of the probe expression installations failed to be requested

Callback Data

ack_cb_fp. The callback function is invoked once for each process for which probe installation is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	all probes were successfully installed in this process
`ASC_operation_failed`	attempt to install probes in this process failed

data_cb_fp. The callback function is invoked once for each message sent from the probe. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback tag is given in the `data_cb_tag` array. The callback message is the data sent by the probe using the `Ais_send` function call.

See Also

`activate_probe, bactivate_probe, bdeactivate_probe, bremove_probe, deactivate_probe, remove_probe`

3.39 load_module

Synopsis

```
#include <Application.h>
AisStatus load_module(
     ProbeMod *module,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

module	probe module to be loaded
ack_cb_fp	callback functin to process load module acknowledgements.
ack_cb_tag	tag to be used as an argument to the callback when it is invoked

Description

This function sends and loads the module from the client side to all the processes within the Application class. Once loaded, the probe expressions available in this probe module can be installed and activated as if those are native in the application.

Note that `load_module` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until the module has been loaded or failed to load within all processes within the application.

Return value

The return value for `load_module` indicates whether the requests to load the indicated module on all processes were successfully submitted. It gives no indication of whether those requests were successfully executed.

`ASC_success`	all load requests were successfully submitted
`ASC_operation_failed`	one or more of the load operations failed to be requested

Callback Data

The callback function is invoked once for each process for which disconnection is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	objects were successfully loaded into this process
`ASC_operation_failed`	attempt to load objects on this process failed

See Also

3.40 operator =

Synopsis

```
#include <Application.h>
Application &operator = (const Application &rhs)
```

Parameters

`rhs`	right operand

Description

This function assigns the value of the right operand to the invoking object. The left operand is the invoking object. For example, "`Application rhs, lhs; ... lhs = rhs;`" assigns the value of `rhs` to `lhs`. Both values would then refer to the same application.

Return value

A reference to the invoking object (i.e., the left operand).

See Also

3.41 remove_phase

Synopsis

```
#include <Application.h>
AisStatus remove_phase(
     Phase ps,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`ps`	phase description to be removed from the application
`ack_cb_fp`	callback function to process phase removal acknowledgments
`ack_cb_tag`	tag to be used as an argument to the acknowledgement callback when it is invoked

Description

This function removes a phase from the application. Data and functions associated with the phase are unaffected by removing the phase. Existing probe data cannot become associated with a phase except at the time of data allocation, so deleting a phase has the effect of permanently disassociating data from any phase.

Note that `remove_phase` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until the phase has been removed or failed to be removed from all processes within the application.

Return value

The return value for `remove_phase` indicates whether the requests to remove the indicated phase on all processes in the application were successfully submitted. It gives no indication of whether the requests were successfully executed.

`ASC_success`	all remove requests were successfully submitted
`ASC_operation_failed`	remove operation failed to be requested to some process

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

ack_cb_fp. The callback function is invoked once for each process for which phase removal is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	phase was successfully removed from this process

`ASC_operation_failed` attempt to remove phase from this process failed

See Also

`add_phase, badd_phase, bremove_phase`

3.42 remove_probe

Synopsis

```
#include <Application.h>
AisStatus remove_probe(
     short count,
     ProbeHandle *phandle,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`count`	number of probe handles in the accompanying array
`phandle`	array of probe handles representing probe expressions to be removed
`ack_cb_fp`	callback function to process probe removal acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function deletes or removes probe expressions that have been installed in an application. If all probe expressions are installed and deactivated, the probe expressions are removed and a "normal" return status results. If one or more of the probe expressions are currently active, the expressions are deactivated and removed and the return status indicates there were active probes at the time of their removal. If one or more of the probes do not exist, all existing probes are removed and the return status indicates an appropriate warning. If one or more of the probe expressions exists but cannot be removed, an error results and none of the probe expressions is removed. If one or more processes are not connected, probe removal takes place within those that are connected, and a warning is issued.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

Probe expression removal is atomic in the sense that all probe expressions are removed from a given process or none are. When probes are removed from a process the process is temporarily suspended, all indicated probes are removed, and the process is resumed. Probe expressions are removed in a process by process basis. There is no synchronization between processes to guarantee that all indicated expressions are removed from all processes. One process may succeed while another one fails.

Note that `remove_probe` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until the probes have been removed or failed to be removed from all processes within the application.

Return value

The return value for `remove_probe` indicates whether the requests to remove the indicated probes on all processes in the application were successfully submitted. It gives no indication of whether the requests were successfully executed.

`ASC_success`	all remove requests were successfully submitted
`ASC_operation_failed`	remove operation failed to be requested to some process

Callback Data

The callback function is invoked once for each process for which probe removal is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	probes were successfully removed from this process
`ASC_operation_failed`	attempt to remove probes from this process failed

See Also

`activate_probe`, `bactivate_probe`, `bdeactivate_probe`, `binstall_probe`, `bremove_probe`, `deactivate_probe`, `install_probe`

3.43 remove_process

Synopsis

```
#include <Application.h>

AisStatus remove_process(int i)
```

Parameters

`i`	position or index into the process table whose entry is to be removed.

Description

This function removes the i^{th} Process object of the application. Parameter `i` must reflect a valid index, that is, that is, 0 `i` < `get_count`(). The process itself is not altered or affected in any way.

The index of a process is not guaranteed to remain invariant when new processes are added to or removed from an application. The index does remain invariant otherwise.

Return value

The return value for `remove_process` indicates whether the process was successfully removed. The return value reflects the highest severity encountered across all processes.

`ASC_success`	process was removed
`ASC_operation_failed`	index was out of bounds

See Also

`add_process, bconnect, bdisconnect, connect, disconnect, get_count`

3.44 resume

Synopsis

```
#include <Application.h>
AisStatus resume(GCBFuncType ack_cb_fp, GCBTagType ack_cb_tag)
```

Parameters

`ack_cb_fp`	callback function to process process resumption acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function resumes execution of an application that has been temporarily suspended by a `suspend` or `bsuspend` function. Execution resumption occurs on a process by process basis. A process must be connected, attached and suspended for it to be resumed. A process that is not connected or not attached will result in a warning return code. A process that is not suspended will result in an informational return code.

Note that `resume` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until the processes have resumed or failed to resume.

Return value

The return value for `resume` indicates whether all requests to resume process execution were successfully submitted. It gives no indication of whether the requests were successfully executed.

`ASC_success`	all request to resume execution were successfully submitted
`ASC_operation_failed`	resume operation failed to be requested for some process

Callback Data

The callback function is invoked once for each process to be resumed. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully resumed
`ASC_operation_failed`	attempt to resume this process failed
ASC_no_sus_res_from_created	must be attached to call resume
ASC_no_sus_res_from_connected	must be attached to call resume

See Also

`attach, battach, bdetach, bresume, bsuspend, detach, suspend`

3.45 send_stdin

Synopsis

```
#include <Application.h>
AisStatus send_stdin(char *buffer, int size)
```

Parameters

`buffer`	character array that contains text to be fed to the application stdin
`size`	number of bytes in the buffer to be given to the application

Description

This function provides text to be used as input to the processies of the application for the `stdin` device, that is, file descriptor 0.

In order for send_stdin to be used, the containted Process objects within the Application must have been created using the create function.

Note that `send_stdin` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the application has received the input.

Return value

The return value for `send_stdin` indicates whether the request to provide application input was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to provide input was successfully submitted
`ASC_operation_failed`	request to provide input failed

Callback Data

The acknowlegement callback function is invoked once for each process in the application when the buffer has been sent to the process. When the callback is invoked, the callback function is passed a pointer to the Process as the callback object. The callback message is the request status, of type `AisStatus`, which may contain one of the status values values that follow.

`ASC_success`	the buffer was successfully sent to poe
`ASC_operation_failed`	attempt to send the buffer to poe failed

See Also

`Process::bcreate, Process::create, PoeAppl::bcreate, PoeAppl::create`

3.46 set_phase_exit

Synopsis

```
#include <Application.h>
AisStatus set_phase_exit(
     Phase ps,
     ProbeExp begin_func,
     GCBFuncType begin_cb_fp,
     GCBTagType begin_cb_tag,
     ProbeExp iter_func,
     GCBFuncType iter_cb_fp,
     GCBTagType iter_cb_tag,
     ProbeExp end_func,
     GCBFuncType end_cb_fp,
     GCBTagType end_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`ps`	phase description to be removed from the application
`begin_func`	initialization function that is executed once within the application when the phase is removed
`begin_cb_fp`	callback function to handle messages from the initialization function
`begin_cb_tag`	tag to be used with the initialization callback function
`iter_func`	iteration function that is executed within the application on each piece of data associated with the phase when the phase is removed
`iter_cb_fp`	callback function to handle messages from the iteration function
`iter_cb_tag`	tag to be used with the iteration callback function
`end_func`	termination function that is executed once within the application when the phase is removed
`end_cb_fp`	callback function to handle messages from the termination function
`end_cb_tag`	tag to be used with the termination callback function
`ack_cb_fp`	callback function to process phase removal acknowledgments

`ack_cb_tag`	tag to be used as an argument to the acknowledgement callback when it is invoked

Description

This function specifies a set of exit functions to be executed when any of the following three events occur.

- when the indicated phase is removed using either the remove_phase or bremove_phase function call
- when diconnecting from the target application (without calling remove_phase or bremove_phase first)
- when the target application has finished execution while the indicated phase is still active

Note that `set_phase_exit` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the exit functions have been placed in the indicated phase or the operation failed to complete.

Each of the phase functions must be loaded into the application before this operation may take place. The function prototypes for the functions are:

- `void begin_func(void *msg_handle)`
- `void iter_func(void *msg_handle, void *data)`
- `void end_func(void *msg_handle)`

Return value

The return value for `remove_phase` indicates whether the requests to remove the indicated phase on all processes in the application were successfully submitted. It gives no indication of whether the requests were successfully executed.

`ASC_success`	all remove requests were successfully submitted
`ASC_operation_failed`	remove operation failed to be requested to some process

Callback Data

begin_cb_fp, **iter_cb_fp**, **end_cb_fp**. These callback functions are invoked each time the corresponding function in the process instrumentation -- `begin_func`, `iter_func`, or `end_func` -- sends a message to the client. The message format is determined by the function that sends the message.

ack_cb_fp. The callback function is invoked once for each process for which phase removal is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	phase was successfully removed from this process
`ASC_operation_failed`	attempt to remove phase from this process failed

See Also

`bset_phase_exit, add_phase, badd_phase, remove_phase, bremove_phase`

3.47 set_phase_period

Synopsis

```
#include <Application.h>
AisStatus set_phase_period(
     Phase ps,
     float period,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`ps`	phase to be modified
`period`	new time interval between successive phase activations, in seconds
`ack_cb_fp`	callback function to process phase acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function changes the time interval between successive activations of a phase. The interval change occurs on a process by process basis for all processes within the application. Processes which do not have the phase installed result in an informational return code. Processes that are not connected result in a warning return code.

The new period is represented by a floating-point value. If the value is positive it represents the time interval in seconds. If the value is zero or positive and smaller than the minimum activation time interval, it represents the minimum activation time interval. In both cases the phase is activated immediately upon setting the new interval. If the value is less than zero the phase is disabled immediately, but left in place for possible future reactivation.

Note that `set_phase_period` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until the phase period has been set or failed to be set within all processes within the application.

Return value

The return value for `set_phase_period` indicates whether all requests to set the phase period were successfully submitted. It gives no indication of whether the requests were successfully executed.

`ASC_success`	all requests to set the phase period were submitted
`ASC_operation_failed`	set phase period failed to be requested for some process

Callback Data

The callback function is invoked once for each process for which setting the new period for a phase is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	phase period was successfully set
`ASC_operation_failed`	attempt to set the phase period on this process failed

See Also

`add_phase, badd_phase, bremove_phase, bset_phase_period, get_phase_period, remove_phase`

3.48 signal - LY

Synopsis

```
#include <Application.h>
AisStatus signal(
     int unix_signal,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`unix_signal`	Unix™ signal to be sent to every process in the application
`ack_cb_fp`	callback function to process signal acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function sends the specified signal to every process in the application. The process must be both connected and attached to receive the signal.

A signal is sent only to those processes that are connected and attached.

Note that `signal` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until processes within the application have been signaled or failed to be signalled.

Return value

The return value for `signal` indicates whether all requests to signal processes were successfully submitted. It gives no indication of whether the requests were successfully executed.

`ASC_success`	all requests to signal the processes were submitted
`ASC_operation_failed`	signalling failed to be requested for some process

Callback Data

The callback function is invoked once for each process for which signalling is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully signaled
`ASC_operation_failed`	attempt to signal this process failed

See Also

3.49 start

Synopsis

```
#include <Application.h>

AisStatus start(GCBFuncType ack_cb_fp, GCBTagType ack_cb_tag)
```

Parameters

`ack_cb_fp`	callback function to process start acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function is currently being designed. This function starts the execution of an application that has been created but not yet begun execution.

Note that `start` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the application has been started or failed to be started.

Return value

The return value for `start` indicates whether the request to start the application was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to start the application was submitted
`ASC_operation_failed`	start failed to be requested

Callback Data

The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	application was successfully started
`ASC_operation_failed`	attempt to start this application failed

See Also

`bstart, PoeAppl::bcreate, PoeAppl::create`

3.50 status

Synopsis

```
#include <Application.h>

AisStatus status(int i)
```

Parameters

`i`	position or index into the process table whose status is to be queried.

Description

This function returns status for the ith Process object of the application. Parameter `i` must reflect a valid index, that is, 0 `i` < `get_count()`. The returned value reflects the status value of the most recently executed blocking call.

Return value

Interpretation of the return value for `status` is determined by the most recent blocking call that was executed.

`ASC_invalid_index`	index does not reflect a valid index

See Also

`get_count`

3.51 suspend

Synopsis

```
#include <Application.h>
AisStatus suspend(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to process suspend acknowledgments
`tag`	tag to be used as an argument to the callback when it is invoked

Description

This function suspends an application that is executing. Application suspension occurs on a process by process basis. A tool must be both connected and attached to a process in order to suspend process execution.

Note that `suspend` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until processes within the application have been suspended or failed to be suspended.

Return value

The return value for `suspend` indicates whether all requests to suspend processes were successfully submitted. It gives no indication of whether the requests were successfully executed.

`ASC_success`	all requests to signal the processes were submitted
`ASC_operation_failed`	signalling failed to be requested for some process

Callback Data

The callback function is invoked once for each process for which suspension is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully suspended
`ASC_operation_failed`	attempt to suspend this process failed
ASC_no_sus_res_from_created	must be attached to call suspend
ASC_no_sus_res_from_connected	must be attached to call suspend

See Also

`attach, battach, bdetach, bresume, bsuspend, detach, resume`

3.52 unload_module

Synopsis

```
#include <Application.h>
AisStatus unload_module(
     ProbeModule *module,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`module`	`probe module to be unloaded.`
`ack_cb_fp`	callback function to process module removal acknowledgments
`ack_cb_tag`	tag to be used as an argument to the acknowledgement callback when it is invoked

Description

This function unloads the module from all the processes within the Application class. Once unloaded, All the probe handles that refer to this probe module are automatically removed.

Note that `unload_module` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until the module has been removed or failed to be removed from all processes within the application.

Return value

The return value for `unload_module` indicates whether the requests to remove the indicated module on all processes were successfully submitted. It gives no indication of whether those requests were successfully executed.

`ASC_success`	all remove requests were successfully submitted
`ASC_operation_failed`	one or more of the remove operations failed to be requested

Callback Data

The callback function is invoked once for each process for which object removal is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	module was successfully removed from this process
`ASC_operation_failed`	attempt to remove module from this process failed

See Also

`bload_module, bunload_module, load_module`

4.0 class GenCallBack

4.1 Supporting Data Types

4.1.1 GCBSysType

Synopsis

```
struct GCBSysType {
     int msg_socket;     // socket over which msg was received
     int msg_type;       // message type
     int msg_size;       // size of the message sent
}
```

Description

This structure is provided as the data type of an input parameter to each callback function as it is invoked. The structure is filled in by the system each time a callback is invoked as the system prepares to invoke the callback.

4.1.2 GCBTagType

Synopsis

```
typedef void *GCBTagType
```

Description

This data type is used by the tag parameter of a callback function. The tag parameter is supplied by the user at the time the callback is registered. Tags are declared as a `void *` to provide adequate space for the tag to be a pointer. The tag itself only has meaning to the callback function and is neither read nor written by the callback system.

4.1.3 GCBObjType

Synopsis

```
typedef void *GCBObjType
```

Description

This data type is used by the object parameter of a callback function. The object parameter is supplied by the system at the time the callback is registered. The object parameter represents a pointer to the object that invokes the asynchronous operation that causes the callback to be invoked. The callback function must know the actual data type of the invoking object and explicitly cast the pointer to be of that type.

4.1.4 GCBMsgType

Synopsis

```
typedef void *GCBMsgType
```

Description

This data type is used by the message parameter of a callback function. The message parameter is supplied by the system at the time the callback is invoked. It is the arrival of this message that causes the callback function to be invoked. The callback function must know the actual data type of the message and explicitly cast the pointer to be of that type.

4.1.5 GCBFuncType

Synopsis

```
typedef void (*GCBFuncType)(
     GCBSysType sys,     // system data structure
     GCBTagType tag,     // user-supplied tag value
     GCBObjType obj,     // object that registers the callback
     GCBMsgType msg)     // activating or invoking message
```

Description

This data type represents a pointer to the callback function. Explicit, user-supplied callback functions are used in all asynchronous function calls.

5.0 class InstPoint

5.1 Supporting Data Types

5.1.1 InstPtLocation

Synopsis

```
#include <InstPoint.h>
enum InstPtLocation {
     IPL_invalid,
     IPL_before,
     IPL_after,
     IPL_replace,
     IPL_LAST_LOCATION
}
```

Description

This enumeration type is used to describe the location of instrumentation relative to the instruction being instrumented. Not all locations are valid with all instrumentation point types. Instrumentation may be placed before the instruction, after the instruction, or the requested code may in some cases replace the instruction in question. Instrumentation points that are not attached to a location within an application or process, perhaps because they were created by a default constructor, are invalid.

5.1.2 InstPtType

Synopsis

```
#include <InstPoint.h>
enum InstPtType {
     IPT_invalid,
     IPT_function_entry,
     IPT_function_exit,
     IPT_function_call,
     IPT_loop_entry,
     IPT_loop_exit,
     IPT_block_entry,
     IPT_block_exit,
     IPT_statement_entry,
     IPT_statement_exit,
     IPT_instruction,
     IPT_LAST_TYPE
}
```

Description

This enumeration type describes the type of location that may be instrumented. Not all will be available within a given source object. Availability depends on source object type and options used when compiling the application process.

See Also

`class SourceObj`

5.2 Constructors

Synopsis

```
#include <InstPoint.h>

InstPoint(void)

InstPoint(const InstPoint &copy)
```

Parameters

`copy`	object to be duplicated in the copy constructor

Description

Two constructors are provided with this class -- a default constructor and a copy constructor. The default constructor is able to create storage, marked as containing invalid instrumentation points, that may later be assigned through an assignment from a valid instrumentation point.

The copy constructor performs a similar operation to assignment, but operates on an uninitialized object.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node

See Also

5.3 get_actuals

Synopsis

```
#include <InstPoint.h>

ProbeExp get_actuals(int i) const
```

Parameters

`i` index of the parameter value to be used

Description

When the instrumentation point refers to a subroutine or function call site, this function returns a reference to the value of the i^{th} parameter of the function being called. When the instrumentation point does not refer to a call site, this function returns an invalid probe expression.

This function returns a reference to the value of the parameter in the call, also known as the *actual parameter*. This is opposed to the *formal parameters* that are given as part of the function definition.

In most cases DPCL cannot know the number or data types of function arguments, so it is incumbent upon the user to be sure the request for a function argument is valid.

Return value

Probe expression referencing the function parameter or marked as invalid.

See Also

`get_type`

5.4 get_container

Synopsis

```
#include <InstPoint.h>

SourceObj get_container(void) const
```

Description

This function returns the source object that contains the instrumentation point. This allows a tool to start with an instrumentation point and explore the context in which it occurs, such as the function and module in which the instrumentation point resides.

Return value

Source object that contains the instrumentation point.

5.5 get_demangled_name

Synopsis

```
#include <InstPoint.h>
char *get_demangled_name(char *buffer, unsigned int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the demangled function name
`len`	maximum number of bytes that will be placed in buffer. The *len* parameter should include enough space for a terminating *null* byte.

Description

When the instrumentation point refers to a subroutine or function call site, this function places the a *null*-terminated string representing the demangled name of the function being called at the location specified by `buffer`. The name may be truncated if the `len` parameter is smaller than the length of the function name.

Return value

Pointer to *buffer*, containing the demangled name of the function

0 if this instrumentation point does not refer to a call site..

See Also

`get_type, get_demangled_name_length`

5.6 get_demangled_name_length

Synopsis

```
#include <InstPoint.h>
unsigned int get_demangled_name_length(void) const
```

Description

This function returns the length, including the terminating *null* byte, of the demangled name of the function being called at this point.

Return value

If this point refers to a function call site, then the length of the demangled name of the function being called.

0 if this point is not a function call site.

See Also

`get_type, get_demangled_name`

5.7 get_line

Synopsis

```
#include <InstPoint.h>
int get_line(void) const
```

Description

This function returns the approximate line number in source where the instrumentation point occurs. If the instrumentation point is invalid, this function returns a value of -1.

Return value

Approximate line number in source or -1.

See Also

5.8 get_location

Synopsis

```
#include <InstPoint.h>
InstPtLocation get_location(void) const
```

Description

This function returns the location of the instrumentation relative to the instrumentation point. Possible locations are: *before*, *after*, *replace*, and *invalid*. If the location is *before*, then instrumentation installed using this instrumentation point will occur immediately before the instruction is executed. If *after*, then instrumentation will be installed immediately after the instruction. If *replace*, the instrumentation will replace the instruction. When the instrumentation point is not attached to a valid location within a process, the return value is *invalid*.

Return value

`IPL_invalid`	instrumentation point is not attached to a valid location
`IPL_before`	instrumentation is placed before the indicated instruction
`IPL_after`	instrumentation is placed after the indicated instruction
`IPL_replace`	instrumentation replaced the indicated instruction

See Also

5.9 get_mangled_name

Synopsis

```
#include <InstPoint.h>
char *get_mangled_name(char *buffer, unsigned int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the mangled function name
`len`	maximum number of bytes that will be placed in buffer. The *len* parameter should include enough space for a terminating *null* byte.

Description

When the instrumentation point refers to a subroutine or function call site, this function places the a *null*-terminated string representing the mangled name (function name with the data type encoded) of the function being called at the location specified by `buffer`. The name may be truncated if the `len` parameter is smaller than the length of the function name.

Return value

Pointer to *buffer*, containing the mangled name of the function

0 if this instrumentation point does not refer to a call site..

See Also

`get_type, get_mangled_name_length`

5.10 get_mangled_name_length

Synopsis

```
#include <InstPoint.h>
unsigned int get_mangled_name_length(void) const
```

Description

This function returns the length, including the terminating *null* byte, of the mangled name of the function being called at this point.

Return value

If this point refers to a function call site, then the length of the mangled name of the function being called.

0 if this point is not a function call site.

See Also

```
get_type, get_mangled_name
```

5.11 get_type

Synopsis

```
#include <InstPoint.h>
InstPtType get_type(void) const
```

Description

This function returns the type of this instrumentation point, such as beginning or end of a subroutine, at a function call site, *etc*.

Return value

Type of instrumentation point.

See Also

5.12 operator =

Synopsis

```
#include <InstPoint.h>
InstPoint &operator = (const InstPoint &copy)
```

Parameters

`copy`	object to be duplicated in the assignment operator

Description

This function copies the argument over the top of the invoking object.

Return value

Reference to the invoking object.

See Also

6.0 Function Group LogSystem

6.1 Supporting Data Types

6.1.1 LoggingDest

Synopsis

```
#include <LogSystem.h>
enum LoggingDest {
     LGD_client,              // info sent to client only
     LGD_daemon,              // info sent to daemon only
     LGD_both,                // info sent to daemon & client
     LGD_neither,             // info is not sent anywhere
}
```

Description

This data type represents ...

6.1.2 LoggingLevel

Synopsis

```
#include <LogSystem.h>
enum LoggingLevel {
     LGL_fatal                // next action is to crash
     LGL_severe               // something is seriously wrong
     LGL_warning              // a warning
     LGL_trace                // function entry/exit
     LGL_detail               // other, more general, info
}
```

Description

This data type represents ...

6.2 Ais_blog_off

Synopsis

```
#include <LogSystem.h>
AisStatus Ais_blog_off(
    const char *hostname)
```

Parameters

Description

Return value

6.3 Ais_blog_on

Synopsis

```
#include <LogSystem.h>
AisStatus Ais_blog_on(
    const char *hostname)

AisStatus Ais_blog_on(
    const char *hostname,
    LoggingLevel level,
    LoggingDest dest,
    GCBFuncType log_cb_fp,
    GCBTagType log_cb_tag)
```

Parameters

Description

Return value

6.4 Ais_log_off

Synopsis

```
#include <LogSystem.h>
AisStatus Ais_log_off(
    const char *hostname,
    GCBFuncType ack_cb_fp,
    GCBTagType ack_cb_tag)
```

Parameters

Description

Return value

6.5 Ais_log_on

Synopsis

```
#include <LogSystem.h>
AisStatus Ais_log_on(
    const char *hostname,
    GCBFuncType ack_cb_fp,
    GCBTagType ack_cb_tag)

AisStatus Ais_log_on(
    const char *hostname,
    LoggingLevel level,
    LoggingDest dest,
    GCBFuncType log_cb_fp,
    GCBTagType log_cb_tag,
    GCBFuncType ack_cb_fp,
    GCBTagType ack_cb_tag)
```

Parameters

Description

Return value

7.0 class Phase

Phases represent the client visible control mechanism for time-initiated instrumentation. In other words, phases are used to control time-sampled instrumentation. Phases are activated, or invoked, when an interval timer expires. The interval timer uses the SIGPROF signal to activate the phase, so applications that use SIGPROF cannot be instrumented with phases.

When a phase is activated it executes its begin function to initialize any data that may be used during the rest of the phase. If the begin function sends any messages back to the client those messages invoke the begin callback function. The begin callback function is invoked once per message sent. After the begin function has completed the data function is then executed, once per datum of probe data associated with the phase. Data is associated with a phase through the `Application::malloc` or `Process::malloc` functions. Any messages sent to the client by the data function are handled on the client by the data callback function. When the data function finishes execution for the last datum, the end function is then executed to perform any necessary clean-up operations. Messages sent by the end function are handled by the end callback.

To fully understand phases it is important to understand that the `Phase` object on the client is a data structure that represents the actual phase. The actual phase resides within the instrumented application process. Certain operations, such as `malloc`, can alter the actual phase in ways that are not reflected within the client data structure. This affects the behavior of the client data structure in subtle ways. In order to provide the most useful abstraction for phases, the default constructor and the copy constructor create new client data structures but they do not create unique phases. As a result, “`Phase p1, p2;`” creates a situation where “`p1 == p2`” is regarded as true. Similarly, the sequence “`Phase p1(f1, f2, t); Phase p2 = p1;`” also results in “`p1 == p2`” evaluating to true. Similar behavior results when the assignment operator, `operator =`, is used.

In contrast, the standard constructors create unique phases even when the parameters used in the constructors are identical. Thus “`Phase p1(f1, f2, t), p2(f1, f2, t);`” results in a situation where “`p1 == p2`” would evaluate to *false* rather than *true*. This possibly counter-intuitive behavior is necessary to allow end-user tools to manage separate groups of data on separate timers.

7.1 Constructors

Synopsis

```
#include <Phase.h>
Phase(void)
Phase(const Phase &copy)

Phase(float period,
     ProbeExp data_func,
     GCBFuncType data_cb,
     GCBTagType data_tg)

Phase(float period,
     ProbeExp begin_func,
     GCBFuncType begin_cb,
     GCBTagType begin_tg,
     ProbeExp data_func,
     GCBFuncType data_cb,
     GCBTagType data_tg,
     ProbeExp end_func,
     GCBFuncType end_cb,
     GCBTagType end_tg)
```

Parameters

`copy`	phase that will be duplicated in a copy constructor
`period`	time interval, in seconds, between successive invocations of the phase
`begin_func`	begin function, executed once upon invocation of the phase
`begin_cb`	begin callback, to which any begin function messages are addressed
`begin_tag`	callback tag for the begin callback `begin_cb`
`data_func`	function that, each time the phase is invoked, is executed once for each datum associated with the phase
`data_cb`	callback function to which any data function messages are addressed
`data_tag`	callback tag for the data function callback `data_cb`

`end_func`	end function, executed once per invocation of the phase after the data function has completed its series of executions
`end_cb`	end callback, to which any end function messages are addressed
`end_tag`	callback tag for the end callback `end_cb`

Description

The default constructor creates an empty phase whose period, functions, callbacks and tags are all set to 0. The default constructor is invoked when uninitialized phases are created, such as in arrays of phases. Objects within the array can be overwritten using an assignment operator (`operator =`).

The copy constructor is used to transfer the contents of an initialized object (the `copy` parameter) to an uninitialized object.

The standard constructors create a new phase and new phase data structure, and initialize the data structure according to the parameters that are provided. The function prototypes are:

- `void begin_func(void *msg_handle)`
- `void data_func(void *msg_handle, void *data)`
- `void end_func(void *msg_handle)`

Exceptions

`ASC_insufficient_memory`	not enough memory to create a new node

See Also

7.2 operator =

Synopsis

```
#include <Phase.h>
```

```
Phase &operator = (const Phase &rhs)
```

Parameters

`rhs` right operand

Description

This function assigns the value of the right operand to the invoking object. The left operand is the invoking object. For example, "`Phase rhs, lhs; ... lhs = rhs;`" assigns the value of `rhs` to `lhs`. Then one can be used interchangeably with the other.

Note that assignment is different from creating two phases using the same input values. For example, "`Phase p1(x,y,z), p2(x,y,z);`" gives two independent phases even though they have exactly the same arguments. Loading `p1` into a process and later unloading `p1` from the same process is, of course, a valid operation. Loading `p1` into a process and later unloading `p2` from the same process as if they were the same phase is invalid, since `p2` represents a different phase with coincidentally the same values.

Return value

A reference to the invoking object (i.e., the left operand).

See Also

7.3 operator ==

Synopsis

```
#include <Phase.h>

int operator == (const Phase &compare)
```

Parameters

`compare` phase to be compared against the invoking object

Description

This function compares two phases for equivalence. If the two objects represent the same phase, this function returns 1. Otherwise it returns 0. For example, “`Phase rhs, lhs; ... lhs = rhs;`” gives a situation where “`rhs == lhs`” is true, and `operator ==` returns 1. But “`Phase p1(x,y,z), p2(x,y,z);`” gives a situation where the value of “`p1 == p2`” is *not* true, even though they were both constructed with the same values, and `operator ==` returns 0.

Return value

This function returns 1 if the two objects are equivalent, 0 otherwise.

See Also

7.4 operator !=

Synopsis

```
#include <Phase.h>
int operator != (const Phase &compare)
```

Parameters

`compare` phase to be compared against the invoking object

Description

This function compares two phases for equivalence. If the two objects represent the same phase, this function returns 0. Otherwise it returns 1. For example, "`Phase rhs, lhs; ... lhs = rhs;`" gives a situation where "`rhs != lhs`" is false, and `operator !=` returns 0. But "`Phase p1(x,y,z), p2(x,y,z);`" gives a situation where the value of "`p1 != p2`" is true, even though they were both constructed with the same values, and `operator !=` returns 1.

Return value

This function returns 0 if the two objects are equivalent, 1 otherwise.

See Also

8.0 class PoeAppl : public Application

The PoeAppl class is derived from the Application class and provides additional convenience functions to provide easier access to poe jobs (MPI programs). These functions can be used to initialize your Application object with the Process objects associated with your MPI program. MPI programs can also be created using the PoeAppl class so that the entire run of the program is available to other DPCL functions.

8.1 Constructors

Synopsis

```
#include <PoeAppl.h>
```

```
PoeAppl(void)
```

Description

Default constructor.

The copy constructor uses the values contained in the copy argument to initialize the new (constructed) object.

Exceptions

Exceptions that could be raised as a result of calling this function are unknown at this time.

8.2 bcreate

Synopsis

```
#include <PoeAppl.h>
      const char *host,
      const char *path,
      const char *args[],
      const char *envp[],
      char *remote_stdin_filename,
      char *remote_stdout_filename,
      char *remote_stderr_filename,
      GCBFuncType stdout_cb_fp,
      GCBTagType stdout_cb_tag,
      GCBFuncType stderr_cb_fp,
      GCBTagType stderr_cb_tag)

AisStatus bcreate(
      const char *host,
      const char *path,
      const char *args[],
      const char *envp[],
      GCBFuncType stdout_cb_fp,
      GCBTagType stdout_cb_tag,
      GCBFuncType stderr_cb_fp,
      GCBTagType stderr_cb_tag)

AisStatus bcreate(
      const char *host,
      const char *path,
      const char *args[],
      const char *envp[],
```

```
        char *remote_stdin_filename,
        char *remote_stdout_filename,
        char *remote_stderr_filename)

AisStatus bcreate(
        const char *host,
        const char *path,
        const char *argv[],
        const char *envp[])
```

Parameters

`host`	host name or IP address of the host machine where the poe application is to be created. This will be the home node for poe.
`path`	complete path to poe, including relative or absolute directory, when appropriate
`argv`	null terminated array of arguments to be provided to poe
`envp`	null terminated array of environment variables to be provided for poe
remote_stdin_filename	remote file to use for stdin
remote_stdout_filename	remote file to use for stdout
remote_stderr_filename	remote file to use for stderr
`stdout_cb_fp`	callback function to handle stdout from the application
`stdout_cb_tag`	tag to be used with the stdout callback function
`stderr_cb_fp`	callback function to handle stderr from the application
`stderr_cb_tag`	tag to be used with the stderr callback function

Description

This function creates an MPI program in a suspended state. All of the processies get created but are suspended at the first executable instruction. Use the start function to allow the MPI program to run.

The poe executable specified in the path parameter is run with the argv and envp provided on the host specified by the host parameter. This will create the MPI program, which will be set up but not run. The configuration of the MPI program will be found and a Process class will be added ro the PoeAppl for each task in the MPI program.

After the create has completed, probe installation, activation, removal, etc. may take place.

To find the number of Processie classes that are now contained in PoeAppl, use Applica tion::get_count. To access a particular process within the PoeAppl, use Applica tion::get_process.

Stdio for the MPI program will be handled by poe depending on the options and environment variables specified for poe. In other words the stdin, stdout and stderr all get funneled through the poe process. The input, output filenames, output callbacks and PoeAppl::send_stdin can be used to access the stdio from and to the poe process.

If you pass callback functions in to the stdout_cb_fp and stderr_cb_fp parameters, the output from poe will be available in these callbacks. Input to poe can be sent using send_stdin().

Another way to access Stdio to poe is to specify the remote filename parameters. In this case stdin, stdout and stderr can be set to use files on the host where poe is running. It is expected that the remote_stdin_filename specified will already exist. The files for the remote_stdin_filename and remote_stdin_filename will created or overwritten if they already exist. If one of the remote file parameters is specified, it takes precedence over the corresponding callback or send_stdin() method of handling Stdio.

Note that `bcreate` does not return control to the caller until the new application has been created or failed to be created. The return value indicates whether the operation succeeded or failed.

Return value

The return value for `bcreate` indicates whether the application was successfully created.

PoeAppl::create is impememted using the existing dpcl interface including calls to Process:create, Process::start to initiate poe on the home node, and PoeAppl::init_procs to initialize the PoeAppl class with contained Process classes. Because of this, return values other than the following may be encountered due to errors in the contained dpcl calls.

`ASC_success`	application was successfully created, as expected
`ASC_operation_failed`	application failed to be created

Callback Data

`stdout_cb_fp`. This callback function is invoked each time the process sends data to `stdout`.

`stderr_cb_fp`. This callback function is invoked each time the process sends data to `stderr`.

The output will be contained in the message parameter of the callback. The size of the output will be contained in the msg_size field of the sys callback parameter. The output from the application may be recieved in different size blocks than were actually sent by the program.

See Also

`bdestroy, bstart, create, destroy, class GenCallback, get_count, get_process, send_stdin, start`

8.3 binit_procs

Synopsis

```
#include <PoeAppl.h>

AisStatus binit_procs(const char *hostname, int poe_pid)
```

Parameters

hostname	A string representing the hostname or ip address where poe was invoked to start the MPI program. This host is referred to as the home node in the poe documentation.
poe_pid	The process identifier (pid) of the poe invocation on its home node.

Description

This call initializes the PoeAppl class to contain the set of processies used by the MPI program. The process and node configuration of the MPI program is read and a corresponding set of Process classes are created containing the pid and hostname of the individual tasks of the MPI program. These Process classes are then added to the PoeAppl.

To find the number of Processie classes that are now contained in PoeAppl, use Application::get_count. To access a particular process within the PoeAppl, use Application::get_process.

A subsequent connect must be issued in order to insert instrumentation into the application.

Note that `binit_procs` does not return control to the caller until either a failure obtaining the MPI program configuration information occurs or all of the Process classes have attempted to be created.

Return value

If the MPI program configuration information is succesfully found and parsed, the return value indicates whether all succeeded or some succeeded and some failed. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

`ASC_success`	processies have been added to PoeAppl class
ASC_bad_att_cfg_version	unrecognized poe version
ASC_bad_att_cfg_numtask	attach config file error parsing number of tasks
ASC_bad_att_cfg_task	attach config file error parsing task number
ASC_bad_att_cfg_ipaddr	attach config file error parsing ip address
ASC_bad_att_cfg_hostname	attach config file error parsing hostname
ASC_bad_att_cfg_pid	attach config file error parsing pid
ASC_bad_att_cfg_sid	attach config file error parsing session id

ASC_bad_att_cfg_progname	attach config file error parsing program name
`ASC_operation_failed`	attempt to connect to this process failed

See Also

`bconnect, connect, get_count, get_process, init_procs`

8.4 create

Synopsis

```
#include <PoeAppl.h>
AisStatus create(
     const char *host,
     const char *path,
     const char *args[],
     const char *envp[],
     char *remote_stdin_filename,
     char *remote_stdout_filename,
     char *remote_stderr_filename,
     GCBFuncType stdout_cb_fp,
     GCBTagType stdout_cb_tag,
     GCBFuncType stderr_cb_fp,
     GCBTagType stderr_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)

AisStatus create(
     const char *host,
     const char *path,
     const char *args[],
     const char *envp[],
     GCBFuncType stdout_cb_fp,
     GCBTagType stdout_cb_tag,
     GCBFuncType stderr_cb_fp,
     GCBTagType stderr_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
AisStatus create(
```

```
        const char *host,
        const char *path,
        const char *args[],
        const char *envp[],
        char *remote_stdin_filename,
        char *remote_stdout_filename,
        char *remote_stderr_filename,
        GCBFuncType ack_cb_fp,
        GCBTagType ack_cb_tag)

    AisStatus create(
        const char *host,
        const char *path,
        const char *args[],
        const char *envp[],
        GCBFuncType ack_cb_fp,
        GCBTagType ack_cb_tag)
```

Parameters

`host`	host name or IP address of the poe process. This will be the hostname sometimes referred to as the home node in the poe documentation.
`path`	complete path to the poe executable, including relative or absolute directory, as appropriate
`args`	null terminated array of arguments to be provided to poe
`envp`	null terminated array of environment variables to be provided to poe
remote_stdin_filename	remote file to use for stdin
remote_stdout_filename	remote file to use for stdout
remote_stderr_filename	remote file to use for stderr
`stdout_cb_fp`	callback function to handle stdout from the application
`stdout_cb_tag`	tag to be used with the stdout callback function
`stderr_cb_fp`	callback function to handle stderr from the application
`stderr_cb_tag`	tag to be used with the stderr callback function

`ack_cb_fp`	callback function to be invoked with a successful or failed creation
`ack_cb_tag`	callback tag to be used when the callback function is invoked

Description

This function creates an MPI program in a suspended state. All of the processies get created but are suspended at the first executable instruction. Use the start function to allow the MPI program to run.

The poe executable specified in the path parameter is run with the argv and envp provided on the host specified by the host parameter. This will create the MPI program, which will be set up but not run. The configuration of the MPI program will be found and a Process class will be added ro the PoeAppl for each task in the MPI program.

After the create has completed, probe installation, activation, removal, etc. may take place.

To find the number of Processie classes that are now contained in PoeAppl, use Applica tion::get_count. To access a particular process within the PoeAppl, use Applica tion::get_process.

Stdio for the MPI program will be handled by poe depending on the options and environment variables specified for poe. In other words the stdin, stdout and stderr all get funneled through the poe process. The input, output filenames, output callbacks and PoeAppl::send_stdin can be used to access the stdio from and to the poe process.

If you pass callback functions in to the stdout_cb_fp and stderr_cb_fp parameters, the output from poe will be available in these callbacks. Input to poe can be sent using send_stdin().

Another way to access Stdio to poe is to specify the remote filename parameters. In this case stdin, stdout and stderr can be set to use files on the host where poe is running. It is expected that the remote_stdin_filename specified will already exist. The files for the remote_stdin_filename and remote_stdin_filename will created or overwritten if they already exist. If one of the remote file parameters is specified, it takes precedence over the corresponding callback or send_stdin() method of handling Stdio.

Note that `create` returns control immediately to the caller. It does not wait until the application has been created. The return value indicates whether the request was successfully submitted and gives no indication whatever about the success or failure of the execution of the request.

Return value

The return value for `create` indicates whether the request to create an application was successfully submitted, but indicates nothing about whether the request was successfully executed.

`ASC_success`	connection was successfully established on this process
`ASC_operation_failed`	attempt to connect to this process failed

Callback Data

The acknowlegement callback function is invoked once when the new application is created. When the callback is invoked the callback function is passed a pointer to the PoeAppl as the callback object. The callback message is the request status, of type `AisStatus`, which may contain one of the status values values that follow.

PoeAppl::create is impememted using the existing dpcl interface including calls to Process:create, Process::start to initiate poe on the home node, and PoeAppl::init_procs to initialize the PoeAppl class with contained Process classes. Because of this, return values other than the following may be encountered due to errors in the contained dpcl calls.

`ASC_success`	connection was successfully established on this process
`ASC_operation_failed`	attempt to connect to this process failed

`stdout_cb_fp`. This callback function is invoked each time the process sends data to `stdout`.

`stderr_cb_fp`. This callback function is invoked each time the process sends data to `stderr`.

The output will be contained in the message parameter of the callback. The size of the output will be contained in the msg_size field of the sys callback parameter. The output from the application may be recieved in different size blocks than were actually sent by the program.

See Also

`bdestroy, bstart, bcreate, destroy, class GenCallback, get_count, get_process, send_stdin, start`

8.5 init_procs

Synopsis

```
#include <PoeAppl.h>
AisStatus init_procs(
     const char *hostname,
     int poe_pid,
     GCBFuncType fp,
     GCBTagType tag)
```

Parameters

hostname	A string representing the hostname or ip address where poe was invoked to start the MPI program. This host is referred to as the home node in the poe documentation.
poe_pid	The process identifier (pid) of the poe invocation on its home node.
fp	callback function to be invoked with a successful or failed initialization of the PoeAppl class.
tag	callback tag to be used as a parameter to the callback when the callback function is invoked.

Description

This call initializes the PoeAppl class to contain the set of processies used by the MPI program. The process and node configuration of the MPI program is read and a corresponding set of Process classes are created containing the pid and hostname of the individual tasks of the MPI program. These Process classes are then added to the PoeAppl.

To find the number of Processie classes that are now contained in PoeAppl, use Application::get_count. To access a particular process within the PoeAppl, use Application::get_process

A subsequent connect must be issued in order to insert instrumentation into the application.

Note that `init_procs` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the configuration has been obtained or for the Process classes to be initialized. The acknowledgement callback function receives notification of the success or failure of the PoeAppl initialization.

Return value

The return value for `init_procs` indicates whether the requests were successfully submitted, but indicates nothing about whether the requests themselves were successfully executed.

`ASC_success`	request to initialize the PoeAppl class was successfully submitted
`ASC_operation_failed`	attempt to submit the request for the MPI program configuration data failed

Callback Data

If the MPI program configuration information is succesfully found and parsed, the return value indicates whether all succeeded or some succeeded and some failed. The function `Application::status(int index)` may be queried to determine whether the operation succeeded or failed on any given process.

`ASC_success`	processies have been added to PoeAppl class
ASC_bad_att_cfg_version	unrecognized poe version
ASC_bad_att_cfg_numtask	attach config file error parsing number of tasks
ASC_bad_att_cfg_task	attach config file error parsing task number
ASC_bad_att_cfg_ipaddr	attach config file error parsing ip address
ASC_bad_att_cfg_hostname	attach config file error parsing hostname
ASC_bad_att_cfg_pid	attach config file error parsing pid
ASC_bad_att_cfg_sid	attach config file error parsing session id
ASC_bad_att_cfg_progname	attach config file error parsing program name
`ASC_operation_failed`	attempt to connect to this process failed

See Also

`bconnect, connect, get_count, get_process, binit_procs`

8.6 send_stdin

Synopsis

```
#include <PoeAppl.h>
AisStatus send_stdin(char *buffer, int size)
```

Parameters

`buffer`	character array that contains text to be fed to the application through the controlling poe process
`size`	number of bytes in the buffer to be sent

Description

This function provides text to be used as input to the poe processs for the `stdin` device, that is, file descriptor 0.

In order for send_stdin to be used, the poe application must have been created using the create function.

Note that `send_stdin` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the application has received the input.

Return value

The return value for `send_stdin` indicates whether the request to provide application input was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to provide input was successfully submitted
`ASC_operation_failed`	request to provide input failed

Callback Data

The acknowlegement callback function is invoked once when the buffer has been sent to poe. When the callback is invoked the callback function is passed a pointer to the PoeAppl as the callback object. The callback message is the request status, of type `AisStatus`, which may contain one of the status values values that follow.

`ASC_success`	the buffer was successfully sent to poe
`ASC_operation_failed`	attempt to send the buffer to poe failed

See Also

`bcreate, create`

9.0 class ProbeExp

Objects of type ProbeExp can be created using the various ProbeExp constructors. Also, there are a few other DPCL objects that can be converted into ProbeExp's.

A SourceObj which represents a variable or a function can be converted to a ProbeExp which represents a reference to the function or variable by using SourceObj::ref_to_probe_exp. A function within a ProbeModule can be converted to a ProbeExp which represents a reference to the function by using ProbeModule::to_probe_exp. An actual parameter of a function being called at the call site of an InstPoint can be converted to a ProbeExp which represents a reference to the parameter by using InstPoint::get_actuals.

9.1 Supporting Data Types

9.1.1 Primitive Data Types

Synopsis

```
typedef char                   int8_t
typedef short                  int16_t
typedef int                    int32_t
typedef long long              int64_t
typedef unsigned char          uint8_t
typedef unsigned short         uint16_t
typedef unsigned int           uint32_t
typedef unsigned long long     uint64_t
typedef float                  float32_t
typedef double                 float64_t
```

Description

This collection of data types represents the primitive data types supported at some level by probe expressions. These are client data types that represent entities used in a probe expression inside an application process. Not all data types are given the same level of support. 32-bit integers are given the greatest level of support, with arithmetic, logical, bitwise, relational and assignment operators. Although pointer values can be manipulated in probe expressions, they are not given a separate data type on the client, but are themselves represented by probe expressions. More complex data types may be allocated for use in probe expressions, but operators that make use of such values are quite limited.

9.1.2 CodeExpNodeType

Synopsis

```
enum CodeExpNodeType {
     CEN_address_op,          // the address of   -- &x
     CEN_and_op,              // bitwise "and"    -- x & y
     CEN_andand_op,           // logical "and"    -- x && y
     CEN_andeq_op,            // bitwise "and"    -- x &= y
     CEN_array_ref_op,        // array reference  -- x[y]
     CEN_call_op,             // function call    -- f(...)
     CEN_div_op,              // division         -- x / y
     CEN_diveq_op,            // divide assign    -- x /= y
     CEN_eq_op,               // assignment       -- x = y
     CEN_eqeq_op,             // value equality   -- x == y
     CEN_ge_op,               // value greater eq -- x >= y
     CEN_gt_op,               // value greater    -- x > y
     CEN_le_op,               // value less or eq -- x <= y
     CEN_lseq_op,             // left shift asgn  -- x <<= y
     CEN_lshift_op,           // left shift       -- x << y
     CEN_lt_op,               // less than        -- x < y
     CEN_minus_op,            // binary minus     -- x - y
     CEN_minuseq_op,          // minus assignment -- x -= y
     CEN_mod_op,              // modulus          -- x % y
     CEN_modeq_op,            // modulus asgn     -- x %= y
     CEN_mult_op,             // multiplication   -- x * y
     CEN_multeq_op,           // multiply asgn    -- x *= y
     CEN_ne_op,               // not equal        -- x != y
     CEN_not_op,              // logical not      -- ! x
     CEN_or_op,               // bitwise or       -- x | y
     CEN_oreq_op,             // bitwise or asgn  -- x |= y
     CEN_oror_op,             // logical or       -- x || y
```

```
CEN_plus_op,             // addition          -- x + y
CEN_pluseq_op,           // addition asgn     -- x += y
CEN_pointer_deref_op,    // pointer deref     -- *x
CEN_postfix_minus_op,    // postfix decr      -- x --
CEN_postfix_plus_op,     // postfix incr      -- x ++
CEN_prefix_minus_op,     // prefix decrement -- -- x
CEN_prefix_plus_op,      // prefix increment -- ++ x
CEN_rseq_op,             // right shift asgn -- x >>= y
CEN_rshift_op,           // right shift       -- x >> y
CEN_tilde_op,            // bitwise negation -- ~ x
CEN_umin_op,             // unary minus       -- - x
CEN_uplus_op,            // unary plus        -- + x
CEN_xor_op,              // exclusive or      -- x ^ y
CEN_xoreq_op,            // exclusive or asgn-- x ^= y
CEN_float32_value,       // float32 value
CEN_float64_value,       // float64 value
CEN_int16_value,         // int16 value
CEN_int32_value,         // int32 value
CEN_int64_value,         // int64 value
CEN_int8_value,          // int8 value
CEN_string_value,        // string value
CEN_uint16_value,        // uint16 value
CEN_uint32_value,        // uint32 value
CEN_uint64_value,        // uint64 value
CEN_uint8_value,         // uint8 value
CEN_if_else_stmt,        // if else      -- if (x) y else z
CEN_if_stmt,             // if stmt           -- if (x) y
CEN_null_stmt,           // null/empty stmt  --
CEN_stmt_list,           // statment list -- x ; y
CEN_undef_node,          // undefined node
```

```
        CEN_stack_ref,           // stack variable
        CEN_heap_ref,            // heap (malloc'd) variable
        CEN_global_variable,     // predefined global variable
        CEN_global_function,     // predefined global function
        CEN_function_ref,        // reference to function
        CEN_actual_param,        // actual parameter
        CEN_LAST_TYPE            // last node type marker
    }
```

Description

The `CodeExpNodeType` enumeration data type represents the various operators and operands that may be found in probe expressions. Probe expressions are structured as *abstract syntax trees*. Expressions are represented with binary operators as a typed node with the left as the left sub-tree, and the right as the right sub-tree.

9.2 Constructors

Synopsis

```
ProbeExp(void)
ProbeExp(int8_t scalar)
ProbeExp(int16_t scalar)
ProbeExp(int32_t scalar)
ProbeExp(int64_t scalar)
ProbeExp(uint8_t scalar)
ProbeExp(uint16_t scalar)
ProbeExp(uint32_t scalar)
ProbeExp(uint64_t scalar)
ProbeExp(float32_t scalar)
ProbeExp(float64_t scalar)
ProbeExp(const char *string)
ProbeExp(const ProbeExp &copy)
```

Parameters

`scalar`	single value of some primitive data type
`string`	null terminated array of signed 8-bit integers, or characters
`copy`	probe expression object that will be duplicated in a copy constructor

Description

All of the above constructors create a new node that may be used as a sub-tree in a larger probe expression. Each of the public constructors, with the exception of the copy constructor, create terminal nodes. To create an expression containing operators one must use the `ProbeExp` operator that corresponds to the desired action. The `ProbeExp` operator constructs the probe expression and performs a validity check. The probe expression may then be installed and activated in an application, at which time additional checks are made to ensure data references are valid within the process.

The copy constructor duplicates the argument, but copies argument children by reference. In other words, it does not duplicate sub-expressions contained as children of `copy`. Instead it duplicates a pointer to the sub-expression and updates the appropriate reference counter.

Exceptions

`ASC_insufficient_memory  not enough memory to create a new node`

9.3 address

Synopsis

```
#include <ProbeExp.h>
ProbeExp address(void) const
```

Description

This function creates a probe expression that represents taking the address of the object in application memory represented by the invoking object. The operand must be an object in application memory. For example, "`ProbeExp exp = obj.address();`" would create an expression `exp` that represents the address of `obj`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Computing the address is valid for any object regardless of data type, but the expression must represent an object in memory. The data type of the result of executing the expression is a pointer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the address of the object represented by the operand.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	invoking object does not represent an object in memory

See Also

9.4 assign

Synopsis

```
#include <ProbeExp.h>

ProbeExp assign(const ProbeExp &rhs) const
```

Parameters

`rhs`	right, or value expression, of the assignment

Description

This function creates an expression where the right operand is evaluated and stored in the location indicated by the left operand. The left operand is represented by the invoking object. For example, "`ProbeExp exp = lhs.assign(rhs);`" would create an expression `exp` that represents evaluating `rhs` and storing its value in the location represented by `lhs`. It is essential that `lhs` represent an object in memory.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the assignment of a value to an object.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of `rhs` (the value assigned) did not match the data type of the invoking object (location assigned to)

See Also

9.5 call

Synopsis

```
#include <ProbeExp.h>
ProbeExp call(short count, ProbeExp *args) const
```

Parameters

`count`	count of arguments or parameters passed to the function being called
`args`	array of arguments or parameters passed to the function being called

Description

This function creates a probe expression that represents a function call. The invoking object represents the function to be called in the application process. For example, the expression "`ProbeExp exp = foo.call(count, args);`" would create an expression `exp` that represents calling a function represented by `foo`. This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing a call to a function.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	one or more arguments to the function does not represent valid a probe expression, either because the expression is ill formed, the expression data type does not match the function argument data type, or data referenced in the expression does not reside on the process

See Also

`ProbeModule::to_probe_exp, SourceObj::ref_to_probe_exp`

9.6 get_data_type

Synopsis

```
#include <ProbeExp.h>
ProbeType get_data_type(void) const
```

Description

This function returns the data type of the probe expression.

Return value

Data type of the probe expression.

See Also

9.7 get_node_type

Synopsis

```
#include <ProbeExp.h>

CodeExpNodeType get_node_type(void) const
```

Description

This function returns the type of node at the root of the probe expression tree. Nodes in a tree represent operators or operands in an executable expression.

Return value

Type of operator or operand at the root of the probe expression tree.

See Also

9.8 has_*

Synopsis

```
int has_int8(void) const
int has_int16(void) const
int has_int32(void) const
int has_int64(void) const
int has_int(void) const
int has_uint8(void) const
int has_uint16(void) const
int has_uint32(void) const
int has_uint64(void) const
int has_uint(void) const
int has_float32(void) const
int has_float64(void) const
int has_float(void) const
int has_string(void) const
int has_name(void) const
int has_text(void) const
int has_children(void) const
int has_left(void) const
int has_right(void) const
int has_center(void) const
```

Description

This family of functions returns a boolean indicator of whether the node being queried represents a datum with the data type in question. Thus `has_int32` will return 1 if the node represents a constant of data type `int32_t`.

Return value

See Also

9.9 ifelse

Synopsis

```
#include <ProbeExp.h>
ProbeExp ifelse(const ProbeExp &te) const
ProbeExp ifelse(const ProbeExp &te, const ProbeExp &ee) const
```

Parameters

`te`	"then" expression, or expression executed when condition is true
`ee`	"else" expression, or expression executed when condition is false

Description

This function creates a probe expression that represents a conditional statement. The invoking object represents the condition to be tested. It must be of type integer or pointer. If the test evaluates to a non-zero value, the expression represented by `te` is executed. If the test evaluates to zero and `ee` is not supplied, execution continues past the conditional. If the test evaluates to zero and `ee` is supplied, then the expression represented by `ee` is executed. For example, "`ProbeExp exp = ce.ifelse(te);`" would create an expression `exp` that represents a conditional statement. The conditional expression to be tested is represented by `ce`, and the expression to be executed should that condition be evaluated to true (any non-zero integer value) is represented by `te`.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing a conditional statement.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of the invoking object is not an integer or pointer

See Also

9.10 is_same_as

Synopsis

```
#include <ProbeExp.h>

int is_same_as(const ProbeExp &compare) const
```

Parameters

`compare`	right hand side of comparison

Description

This function compares two probe expressions for equivalence. If the invoking object has the same structure as the probe expression it is compared against, this function returns 1. If the structure is different in some way, or the expressions are similar in structure but have different values at corresponding nodes, it returns 0.

Return value

This function returns 1 when the expressions are equivalent, otherwise 0.

See Also

9.11 operator + (binary)

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator + (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the addition of two operands. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs + rhs;`" would create an expression `exp` that represents the addition of two values, `lhs` and `rhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Addition is only valid when both operands are integers, or one operand is an integer and one is a pointer. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer. When one operand is a pointer, it has the usual meaning associated with pointer arithmetic as defined in C/C++, and the data type associated with the result is a pointer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the addition of two operands.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.12 operator + (unary)

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator + (void) const
```

Description

This function is effectively a no-op. It simply returns the value of its operand.

Return value

Probe expression representing the left operand.

Exceptions

`ASC_insufficient_memory` insufficient memory to create a new node

See Also

9.13 operator +=

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator += (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the addition of two operands, and its subsequent storage of the result into the invoking object. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, the expression "`ProbeExp exp = lhs += rhs;`" would create an expression `exp` that represents the addition of two values, `lhs` and `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Addition is only valid when both operands are integers, or the left operand is a pointer and the right operand is an integer. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer. When `lhs` is a pointer, it has the usual meaning associated with pointer arithmetic as defined in C/C++ and the data type of the result of executing the expression is a pointer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the addition of two operands and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.14 operator ++ (prefix)

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator ++ (void) const
```

Description

This function creates a probe expression that represents the increment of an integer operand. The operand is the invoking object. The operand must be an expression that represents an object in memory. The result of the operation is the value of the operand after the increment takes place. For example, "`ProbeExp exp = ++rhs;`" would create an expression `exp` that represents incrementing `rhs` by one. The expression `exp` could then be used as a subexpression in an assignment or other type of statement or expression.

Increment is only valid when the operand is a signed integer or a pointer. Any other operand data type is invalid. When the operand is an integer it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer. When `rhs` is a pointer, it has the usual meaning associated with pointer arithmetic as defined in C/C++ and the data type of the result of executing the expression is a pointer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the addition of one to an operand and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of the operand is inappropriate

See Also

9.15 operator ++ (postfix)

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator ++ (int zero) const
```

Parameters

`zero`	constant integer zero

Description

This function creates a probe expression that represents the increment of an integer operand. The operand is the invoking object. The operand must be an expression that represents an object in memory. The result of the operation is the value of the operand before the increment takes place. For example, "`ProbeExp exp = lhs++;`" would create an expression `exp` that represents incrementing `lhs` by one. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Increment is only valid when the operand is a signed integer or a pointer. Any other operand data type is invalid. When the operand is an integer it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer. When `lhs` is a pointer, it has the usual meaning associated with pointer arithmetic as defined in C/C++ and the data type of the result of executing the expression is a pointer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the addition of one to an operand and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of the operand is inappropriate

See Also

9.16 operator - (binary)

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator - (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the subtraction of two operands. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, “`ProbeExp exp = lhs - rhs;`” would create an expression `exp` that represents the subtraction of `rhs` from `lhs`. The expression `exp` could then be used as a subexpression in an assignment or other type of statement or expression.

Subtraction is only valid when both operands are integers, or the left operand is a pointer and the right operand is an integer, or both operands are pointers of the same type. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer. When one or both operand is a pointer, it has the usual meaning associated with pointer arithmetic as defined in C/C++, and the data type associated with the result is a pointer. When both operands are pointers, it has the usual meaning associated with pointer subtraction as defined in C/C++, and the data type associated with the result is a signed integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the subtraction of two operands.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.17 operator - (unary)

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator - (void) const
```

Description

This function creates a probe expression that represents the arithmetic negation of an operand. The right operand represents the invoking object. The operand may be an object in memory or an expression that evaluates to a value. For example, "`ProbeExp exp = - rhs;`" would create an expression `exp` that represents the negation of `rhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Negation is only valid when the operand is a signed integer. Any other operand data type is invalid. When the operand is an integer it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the arithmetic negation of an operand.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of the operand is inappropriate

See Also

9.18 operator -=

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator -= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the subtraction of two operands, and its subsequent storage of the result into the invoking object. The left operand represents the invoking object, while the argument `rhs` represents the right operand. The left operand must be an object in memory, while the right operand may be an object in memory or an expression that evaluate to a value. For example, “`ProbeExp exp = lhs -= rhs;`” would create an expression `exp` that represents the subtraction of two values, `lhs` and `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Subtraction is only valid when both operands are integers, or the left operand is pointer and the right operand is an integer. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer. When `lhs` is a pointer, it has the usual meaning associated with pointer arithmetic as defined in C/C++ and the data type of the result of executing the expression is a pointer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the subtraction of two operands and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.19 operator -- (prefix)

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator -- (void) const
```

Description

This function creates a probe expression that represents the decrement of an integer operand. The operand is the invoking object. The operand must be an expression that represents an object in memory. The result of the operation is the value of the operand after the decrement takes place. For example, "`ProbeExp exp = --rhs;`" would create an expression `exp` that represents decrementing `rhs` by one. The expression `exp` could then be used as a subexpression in an assignment or other type of statement or expression.

Decrement is only valid when the operand is a signed integer or a pointer. Any other operand data type is invalid. When the operand is an integer it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer. When `rhs` is a pointer, it has the usual meaning associated with pointer arithmetic as defined in C/C++ and the data type of the result of executing the expression is a pointer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the subtraction of one from an operand and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of the operand is inappropriate

See Also

9.20 operator -- (postfix)

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator -- (int zero) const
```

Parameters

`zero`	constant integer zero

Description

This function creates a probe expression that represents the decrement of an integer operand. The operand is the invoking object. The operand must be an expression that represents an object in memory. The result of the operation is the value of the operand before the decrement takes place. For example, "`ProbeExp exp = lhs--;`" would create an expression `exp` that represents decrementing `lhs` by one. The expression `exp` could then be used as a subexpression in an assignment or other type of statement or expression.

Decrement is only valid when the operand is a signed integer or a pointer. Any other operand data type is invalid. When the operand is an integer it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer. When `lhs` is a pointer, it has the usual meaning associated with pointer arithmetic as defined in C/C++ and the data type of the result of executing the expression is a pointer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the subtraction of one from an operand and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of the operand is inappropriate

See Also

9.21 operator * (binary)

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator * (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the multiplication of two operands. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs * rhs;`" would create an expression `exp` that represents the multiplication of `rhs` by `lhs`. The expression `exp` could then be used as a subexpression in an assignment or other type of statement or expression.

Multiplication is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the multiplication of two operands.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.22 operator * (unary)

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator * (void) const
```

Description

This function creates a probe expression that represents the dereferencing of a pointer operand. The right operand represents the invoking object. The operand may be an object in memory or an expression that evaluates to a value. For example, “`ProbeExp exp = * rhs;`” would create an expression `exp` that represents the object pointed to by the pointer value `rhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Pointer dereferenceing is only valid when the operand is a pointer. Any other operand data type is invalid. When the operand is a pointer it has the usual meaning associated with dereferencing pointers and the data type of the result of executing the expression is the data type of the pointee.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the dereferencing of a pointer operand.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of the operand is inappropriate

See Also

9.23 operator *=

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator *= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the multiplication of two operands, and its subsequent storage of the result into the invoking object. The left operand represents the invoking object, while the argument `rhs` represents the right operand. The left operand must be an object in memory, while the right operand may be an object in memory or an expression that evaluates to a value. For example, “`ProbeExp exp = lhs *= rhs;`” would create an expression `exp` that represents the multiplication of two values, `lhs` and `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Multiplication is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the multiplication of two operands and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.24 operator /

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator / (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the division of two operands. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs / rhs;`" would create an expression `exp` that represents the division of `rhs` by `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Division is only valid when both operands are integers, and the divisor is non-zero. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the division of two operands.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.25 operator /=

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator /= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the division of two operands, and its subsequent storage of the result into the invoking object. The left operand represents the invoking object, while the argument `rhs` represents the right operand. The left operand must be an object in memory, while the right operand may be an object in memory or an expression that evaluates to a value. For example, "`ProbeExp exp = lhs /= rhs;`" would create an expression `exp` that represents the division of two values, `lhs` and `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Division is only valid when both operands are integers, and the divisor is non-zero. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the division of two operands and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.26 operator %

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator % (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the division of two operands, where the remainder rather than the dividend is returned. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, “`ProbeExp exp = lhs % rhs;`” would create an expression `exp` that represents the division of `rhs` by `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Division is only valid when both operands are integers, and the divisor is non-zero. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the remainder of the division of two operands.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.27 operator %=

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator %= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents the division of two operands, where the remainder rather than the dividend is returned, and its subsequent storage of the result into the invoking object. The left operand represents the invoking object, while the argument `rhs` represents the right operand. The left operand must be an object in memory, while the right operand may be an object in memory or an expression that evaluates to a value. For example, "`ProbeExp exp = lhs %= rhs;`" would create an expression `exp` that represents the division of two values, `lhs` and `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Division is only valid when both operands are integers, and the divisor is non-zero. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with computer arithmetic of signed integers and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the division of two operands and assignment of the remainder.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.28 operator =

Synopsis

```
#include <ProbeExp.h>
ProbeExp &operator = (const ProbeExp &rhs)
```

Parameters

`rhs` right operand

Description

This function does *not* create a node in a probe expression tree. Rather, it performs a local assignment on the client, of the value in the right operand to the object represented by the left operand. For example, "`ProbeExp lhs; lhs = rhs;`" would assign the value contained in `rhs` to the variable `lhs`. Notice that the above example is *different* from "`ProbeExp lhs = rhs;`" in that the first example invokes the assignment operator, "`operator =`", while the second example invokes the copy constructor. But though different functions are called the end result is the same, that is, the probe expression represented by the right operand is assigned to the object represented by the left operand.

Return value

A reference to the invoking object (i.e., the left operand).

See Also

9.29 operator ==

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator == (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a comparison for equality of two operands, where 1 is returned if they are equal, and 0 is returned if they are not. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs == rhs;`" would create an expression `exp` that represents a comparison for equality of `rhs` and `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Comparison for equality is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with comparison of signed integers and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the comparison of two operands for equality.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.30 operator !

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator ! (void) const
```

Description

This function creates a probe expression that represents the logical negation of an operand, where 0 is returned if the operand is a non-zero value, and 1 is returned if the operand is 0. The right operand represents the invoking object. The operand may be an object in memory or an expression that evaluates to a value. For example, "`ProbeExp exp = ! rhs;`" would create an expression `exp` that represents the negation of `rhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Logical negation is only valid when the operand is an integer, a pointer, or an actual parameter. Any other operand data type is invalid. The operator has the usual meaning associated with computer logic, and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the negation of an operand.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of the operand is inappropriate

See Also

9.31 operator !=

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator != (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a comparison for inequality of two operands, where 0 is returned if they are equal, and 1 is returned if they are not. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, “`ProbeExp exp = lhs != rhs;`” would create an expression `exp` that represents a comparison for equality of `rhs` and `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Comparison for equality is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with comparison of signed integers and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the comparison of two operands for inequality.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.32 operator <

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator < (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a comparison of two operands, where 1 is returned if the left operand is less than the right operand, and 0 is returned otherwise. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs < rhs;`" would create an expression `exp` that represents a comparison of `rhs` and `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Comparison is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with relational operators and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the comparison of two operands for relative size.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.33 operator <=

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator <= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a comparison of two operands, where 1 is returned if the left is less than or equal to the right, and 0 is returned otherwise. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs <= rhs;`" would create an expression `exp` that represents a comparison of `rhs` and `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Comparison is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with relational operators and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the comparison of two operands for relative size.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.34 operator <<

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator << (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise left shift of the left operand. When the right operand is positive, the value returned is the left operand shifted that many places to the left. When the right operand is zero, the value returned is the value of the left operand. When the right operand is negative, the shift operation is not defined, and the value returned is unpredicatable. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs << rhs;`" would create an expression `exp` that represents a left shift of `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Left shift is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise shift operators and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the left shift of the left operator.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.35 operator <<=

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator <<= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise left shift of the left operand. When the right operand is positive, the value returned is left operand shifted that many places to the left. When the right operand is zero, the value returned is the value of the left operand. When the right operand is negative, the shift operation is not defined, and the value returned is unpredicatable. The result is subsequently stored into the invoking object. The left operand represents the invoking object, while the argument `rhs` represents the right operand. The left operand must be an object in memory, while the right operand may be an object in memory or an expression that evaluates to a value. For example, "`ProbeExp exp = lhs <<= rhs;`" would create an expression `exp` that represents the left shift of `lhs` by `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Shift operations are only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise shift operations and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing a left bit-wise shift and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.36 operator >

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator > (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a comparison of two operands, where 1 is returned if the left operand is greater than the right operand, and 0 is returned otherwise. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs > rhs;`" would create an expression `exp` that represents a comparison of `rhs` and `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Comparison is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with relational operators and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the comparison of two operands for relative size.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.37 operator >=

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator >= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a comparison of two operands, where 1 is returned if the left is greater than or equal to the right, and 0 is returned otherwise. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs >= rhs;`" would create an expression `exp` that represents a comparison of `rhs` and `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Comparison is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with relational operators and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the comparison of two operands for relative size.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.38 operator >>

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator >> (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise right shift of the left operand. When the right operand is positive, the value returned is the left operand shifted that many places to the right. When the right operand is zero, the value returned is the value of the left operand. When the right operand is negative, the shift operation is not defined, and the value returned is unpredicatable. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs >> rhs;`" would create an expression `exp` that represents a left shift of `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Right shift is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise shift operators and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the right shift of the left operator.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.39 operator >>=

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator >>= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise right shift of the left operand. When the right operand is positive, the value returned is left operand shifted that many places to the right. When the right operand is zero, the value returned is the value of the left operand. When the right operand is negative, the shift operation is not defined, and the value returned is unpredicatable. The result is subsequently stored into the invoking object. The left operand represents the invoking object, while the argument `rhs` represents the right operand. The left operand must be an object in memory, while the right operand may be an object in memory or an expression that evaluates to a value. For example, "`ProbeExp exp = lhs >>= rhs;`" would create an expression `exp` that represents the right shift of `lhs` by `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Shift operations are only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise shift operations and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing a right bit-wise shift and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.40 operator & (binary)

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator & (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise *AND* of the left and right operands. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs & rhs;`" would create an expression `exp` that represents a bit-wise *AND* of `lhs` and `rhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Bit-wise *AND* is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise *AND* operators and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the bit-wise *AND* of the left and right operands..

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.41 operator & (unary)

Synopsis

```
#include <ProbeExp.h>

ProbeExp *operator & (void)
```

Description

This function does *not* create a node in a probe expression tree. Rather, it computes and returns the address of the invoking object on the client. For example, the probe expression "`ProbeExp *ptr = &obj;`" would store a pointer to the object `obj` in the pointer `ptr`. It is necessary that the function work in this manner and *not* create an expression tree, to allow C++ to pass objects by reference.

Return value

A pointer to the invoking object on the client.

See Also

9.42 operator &=

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator &= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise *AND* of the operands. The result is subsequently stored into the invoking object. The left operand represents the invoking object, while the argument `rhs` represents the right operand. The left operand must be an object in memory, while the right operand may be an object in memory or an expression that evaluates to a value. For example, "`ProbeExp exp = lhs &= rhs;`" would create an expression `exp` that represents the bit-wise *AND* of `lhs` and `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Bit-wise operations are only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise *AND* operations and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing a bit-wise *AND* and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.43 operator &&

Synopsis

```
#include <ProbeExp.h>
```

```
ProbeExp operator && (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a logical *AND* of two operands, where 1 is returned both operands are non-zero, and 0 is returned if one or more are not. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs && rhs;`" would create an expression `exp` that represents a logical *AND* of `rhs` and `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Logical *AND* is only valid when each operands is an integer, a pointer, or an actual parameter. Any other combination of operand data types is invalid. The operator has the usual meaning associated with logical expressions, and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the logical *AND* of two operands.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.44 operator |

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator | (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise *OR* of the left and right operands. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs | rhs;`" would create an expression `exp` that represents a bit-wise *OR* of `lhs` and `rhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Bit-wise *OR* is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise *OR* operators and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the bit-wise *OR* of the left and right operands..

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.45 operator |=

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator |= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise *OR* of the operands. The result is subsequently stored into the invoking object. The left operand represents the invoking object, while the argument `rhs` represents the right operand. The left operand must be an object in memory, while the right operand may be an object in memory or an expression that evaluates to a value. For example, "`ProbeExp exp = lhs |= rhs;`" would create an expression `exp` that represents the bit-wise *OR* of `lhs` and `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Bit-wise operations are only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise *OR* operations and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing a bit-wise *OR* and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.46 operator ||

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator || (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a logical *OR* of two operands, where 1 is returned at least one operand is non-zero, and 0 is returned if both are zero. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs || rhs;`" would create an expression `exp` that represents a logical *OR* of `rhs` and `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Logical *OR* is only valid when each operands is an integer, a pointer, or an actual parameter. Any other combination of operand data types is invalid. The operator has the usual meaning associated with logical expressions, and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the logical *OR* of two operands.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.47 operator ^

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator ^ (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise *exclusive-OR* of the left and right operands. The invoking object represents the left operand, while the argument `rhs` represents the right operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs ^ rhs;`" would create an expression `exp` that represents a bit-wise *exclusive-OR* of `lhs` and `rhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Bit-wise *exclusive-OR* is only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise *exclusive-OR* operators and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the bit-wise *exclusive-OR* of the left and right operands..

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is not an integer

See Also

9.48 operator ^=

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator ^= (const ProbeExp &rhs) const
```

Parameters

`rhs`	right operand

Description

This function creates a probe expression that represents a bit-wise *exclusive-OR* of the operands. The result is subsequently stored into the invoking object. The left operand represents the invoking object, while the argument `rhs` represents the right operand. The left operand must be an object in memory, while the right operand may be an object in memory or an expression that evaluates to a value. For example, “`ProbeExp exp = lhs ^= rhs;`” would create an expression `exp` that represents the bit-wise *exclusive-OR* of `lhs` and `rhs`, and its assignment to `lhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Bit-wise operations are only valid when both operands are integers. Any other combination of operand data types is invalid. When both operands are integers it has the usual meaning associated with bit-wise *exclusive-OR* operations and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing a bit-wise *exclusive-OR* and assignment of the result.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.49 operator ~

Synopsis

```
#include <ProbeExp.h>

ProbeExp operator ~ (void) const
```

Description

This function creates a probe expression that represents the bit-wise inversion of an operand. The right operand represents the invoking object. The operand may be an object in memory or an expression that evaluates to a value. For example, "`ProbeExp exp = ~ rhs;`" would create an expression `exp` that represents the inversion of `rhs`. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Bit-wise inversion is only valid when the operand is a signed integer. Any other operand data type is invalid. When the operand is an integer it has the usual meaning associated with computer logic and the data type of the result of executing the expression is an integer.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the bit-wise inversion of an operand.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of the operand is inappropriate

See Also

9.50 operator []

Synopsis

```
#include <ProbeExp.h>
ProbeExp operator [] (int index) const
ProbeExp operator [] (ProbeExp index) const
```

Parameters

`index`	index into the array or pointer offset

Description

This function creates a probe expression that represents the indexing and dereference of a pointer operand. The invoking object represents the left (pointer) operand, while the argument `index` represents the right (index) operand. The operands may be objects in memory or expressions that evaluate to values. For example, "`ProbeExp exp = lhs [ rhs ];`" would create an expression `exp` that represents adding `rhs` to `lhs` and dereferencing the result. The expression `exp` could then be used as a sub-expression in an assignment or other type of statement or expression.

Index and dereference is only valid when the left operand is a pointer and the right operand is an integer. Any other combination of operand data types is invalid. When both operands are of appropriate data types it has the usual meaning associated with index and dereferencing and the data type of the result of executing the expression matches the pointee.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the index and dereference of the left and right operands.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node
`ASC_invalid_espression`	data type of one or both operands is inappropriate

See Also

9.51 sequence

Synopsis

```
#include <ProbeExp.h>
ProbeExp sequence(const ProbeExp &second) const
```

Parameters

`second`	second expression in the sequence

Description

This function creates a probe expression that represents the joining of two probe expressions into a sequence. The invoking object represents the first expression in the sequence to be executed, while the argument `second` represents the second expression to be executed. The operands may be objects in memory or expressions that evaluate to values. For example, “`ProbeExp exp = first.sequence(second);`” would create an expression `exp` that represents the execution of `first` followed by `second`. The expression `exp` could then be used as a sub-expression in a conditional expression, a sequence, or other type of statement or expression.

This expression may be executed on the application process only after it has been installed and activated.

Return value

Probe expression representing the sequencing of two expressions.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node

See Also

9.52 value_*

Synopsis

```
int8_t value_int8(void) const
int16_t value_int16(void) const
int32_t value_int32(void) const
int64_t value_int64(void) const
uint8_t value_uint8(void) const
uint16_t value_uint16(void) const
uint32_t value_uint32(void) const
uint64_t value_uint64(void) const
float32_t value_float32(void) const
float64_t value_float64(void) const
ProbeExp value_left(void) const
ProbeExp value_right(void) const
ProbeExp value_center(void) const
```

Description

Returns the value contained in the node.

Return value

The value, of the indicated type, contained within the node.

Exceptions

`ASC_invalid_value_ref`	node does not contain a value of the indicated type

See Also

`value_text, value_text_length`

9.53 value_text

Synopsis

```
char *value_text(char *buffer, unsigned int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the text value
`len`	maximum number of bytes the function will place in `buffer`. The `len` parameter should include enough space for a terminating *null* byte.

Description

Copies into *buffer* a *null*-terminated string representing the value contained within the node. The value may be truncated if the `len` parameter is smaller than the length of the text value.

Return value

A pointer to `buffer`, which will contain at most `len` bytes of the text value contained within the node.

Exceptions

`ASC_invalid_value_ref`	node does not contain a value of the indicated type

See Also

`value_*, value_text_length`

9.54 value_text_length

Synopsis

```
unsigned int value_text_length(void) const
```

Description

Returns the length, including the terminating *null* byte, of the text value contained within the node.

Return value

The length of the text value contained within the node.

Exceptions

`ASC_invalid_value_ref`	node does not contain a value of the indicated type

See Also

`value_*, value_text`

10.0 class ProbeHandle

10.1 Constructors

Synopsis

```
#include <ProbeHandle.h>

ProbeHandle(void)

ProbeHandle(const ProbeHandle &copy)
```

Parameters

`copy`	object to be duplicated in the copy constructor

Description

Two constructors are provided with this class -- a default constructor and a copy constructor. The default constructor is able to create storage, marked initially as containing invalid probe handles, that may later be assigned or initialized through a probe installation.

The copy constructor performs a similar operation to assignment, but operates on an uninitialized object.

Exceptions

`ASC_insufficient_memory`	insufficient memory to create a new node

See Also

10.2 get_expression

Synopsis

```
#include <ProbeHandle.h>

ProbeExp get_expression(void)
```

Description

This function returns the original probe expression installed in the application process. Note that the expression returned is the original and not a copy, so alterations to the original after it has been installed will be reflected in the the expression returned by this function.

Return value

Original probe expression installed in the application process.

See Also

10.3 get_point

Synopsis

```
#include <ProbeHandle.h>

InstPoint get_point(void)
```

Description

This function returns the original instrumentation point where the probe expression was installed in the application process.

Return value

Instrumentation point where the probe expression was installed in the application process.

See Also

10.4 operator =

Synopsis

```
#include <ProbeHandle.h>
ProbeHandle &operator = (const ProbeHandle &copy)
```

Parameters

`copy`	object to be duplicated in the assignment operator

Description

This function copies the argument over the top of the invoking object.

Return value

Reference to the invoking object.

See Also

11.0 class ProbeModule

ProbeModule is an object file resides on the client side that can be dynamically loaded into the application process. One can build a complex probe expression from this module but the probe module must be loaded before one can install and activate this probe expression.

UnderAIX, one can write a ProbeModule using C language (C++ is not supported). The source file needs to be compiled and only those exported functions are visible from the ProbeModule. That is, those function names are put in an exported file and invoke linker with -bE: flag.. A complete makefile example can be found in /usr/lpp/ppe.dpcl/samples/probe_module

11.1 Constructors

Synopsis

```
#include <ProbeModule.h>

ProbeModule(void)

ProbeModule(const ProbeModule &copy)

ProbeModule(const char *filename)
```

Parameters

`copy`	probe module that will be duplicated in a copy constructor
`filename`	name and path of an object file (*.o) that contains functions to be loaded into the application process

Description

The default constructor creates an empty probe module structure, in other words, a structure that contains no objects. The default constructor is invoked when uninitialized probe modules are created, such as in arrays. Objects within the array can be overwritten using an assignment operator (`operator =`).

The copy constructor is used to transfer the contents of an initialized object (the `copy` parameter) to an uninitialized object.

The standard constructor reads the object file (*.o) that contains functions to be loaded into the application process. It reads the file to determine what functions are available and the data type signature of each.

Exceptions

ASC_insufficient_memory	not enough memory to create a new node
ASC_module_invalid	invalid probe module
ASC_module_not_found	the module cannot be found

See Also

`load_module, bload_module, unload_module, bunload_module`

11.2 get_count

Synopsis

```
#include <ProbeModule.h>

int get_count(void) const
```

Description

This function returns the number of functions in the module. If the module was initialized by a default constructor or its value was copied from a default constructor, this function returns 0.

Return value

Number of functions in the module, or 0 if the module was initialized by a default constructor.

See Also

11.3 get_name

Synopsis

```
#include <ProbeModule.h>

char *get_name(int index, char *buffer, unsigned int len) const
```

Parameters

`index`	index of the desired function, equal to or greater than zero, and less than `get_count()`
`buffer`	caller-allocated buffer to hold the module name
`len`	maximum number of bytes the function will place in buffer. The *len* parameter should include enough space for a terminating *null* byte.

Description

This function copies into *buffer* a *null*-terminated string representing the name of the desired function. The name may be truncated if the *len* parameter is smaller than the length of the name. If the index is out of range, that is, if it is less than zero or equal to or greater than `get_count()`, it returns 0.

Return value

A pointer to *buffer*, which will contain at most *len* bytes of the name of the desired function, or 0 if the index is out of range.

See Also

`get_name_length`

11.4 get_name_length

Synopsis

```
#include <ProbeModule.h>
unsigned int get_name_length(int index) const
```

Parameters

`index`	index of the desired function, equal to or greater than zero, and less than `get_count()`

Description

This function returns the length, including the terminating *null* byte, of the (mangled) name of the desired function. If the index is out of range, that is, if it is less than zero or equal to or greater than `get_count()`, it returns 0.

Return value

The length of the name of the desired function, or 0 if the index is out of range.

See Also

`get_name`

11.5 operator =

Synopsis

```
#include <ProbeModule.h>

ProbeModule &operator = (const ProbeModule &rhs)
```

Parameters

`rhs`	right operand

Description

This function assigns the value of the right operand to the invoking object. The left operand is the invoking object. For example, "`ProbeModule rhs, lhs; ... lhs = rhs;`" assigns the value of `rhs` to `lhs`. Then one can be used interchangeably with the other.

Return value

A reference to the invoking object (i.e., the left operand).

See Also

11.6 operator ==

Synopsis

```
#include <ProbeModule.h>
int operator == (const ProbeModule &compare) const
```

Parameters

`compare` probe module to be compared against the invoking object

Description

This function compares two probe modules for equivalence. If the two objects represent the same probe module, this function returns 1. Otherwise it returns 0.

Return value

This function returns 1 if the two objects are equivalent, 0 otherwise.

See Also

11.7 operator !=

Synopsis

```
#include <ProbeModule.h>

int operator != (const ProbeModule &compare) const
```

Parameters

`compare` probe module to be compared against the invoking object

Description

This function compares two probe modules for equivalence. If the two objects represent the same probe module, this function returns 0. Otherwise it returns 1.

Return value

This function returns 0 if the two objects are equivalent, 1 otherwise.

See Also

11.8 to_probe_exp

Synopsis

```
#include <ProbeModule.h>

ProbeExp to_probe_exp(int index) const
```

Parameters

`index` index of the desired function , equal to or greater than zero, and less than `get_count()`

Description

This function returns a probe expression that represents a reference to the desired function. The probe expression may be used to form a call to that function. If the index is out of range, that is, if it is less than zero or equal to or greater than `get_count()`, it returns an "undefined" probe expression.

Return value

A probe expression that represents a referenct to the desired function, or "undefined" if the index is out of range.

See Also

12.0 class ProbeType

12.1 Supporting Data Types

12.1.1 DataExpNodeType

Synopsis

```
enum DataExpNodeType {
     DEN_array_type,     // array type decl  -- x[y]
     DEN_class_type,     //
     DEN_enum_type,      // enum type decl   -- enum x {y}
     DEN_float32_type,   // float32 type decl
     DEN_float64_type,   // float64 type decl
     DEN_function_type,  //
     DEN_int16_type,     // int16 type declaration
     DEN_int32_type,     // int32 type declaration
     DEN_int64_type,     // int64 type declaration
     DEN_int8_type,      // int8 type declaration
     DEN_pointer_type,   // pointer type exp -- * x
     DEN_reference_type, // reference type   -- & x
     DEN_struct_type,    //
     DEN_uint16_type,    // uint16 type declaration
     DEN_uint32_type,    // uint32 type declaration
     DEN_uint64_type,    // uint64 type declaration
     DEN_uint8_type,     // uint8 type declaration
     DEN_union_type,     //
     DEN_user_type,      // user defined type name
     DEN_void_type,      // void data type
     DEN_default_type,   // default constructor type
     DEN_unspecified_type,// has size but no structure
     DEN_error_type,     // result of failed operation
```



```
    DEN_LAST_TYPE
}
```

Description

Values of type `ProbeType` are expression trees that represent the data type of an object within an application process. The object may be an application object, that is, it may be a part of the application program, or it may be a probe object, that is, an object allocated and used by the instrumentation system. This data structure reflects all of the possible enumeration values used by the expression tree to represent the data type of the object. It is a combination of the enumeration value of each node, and the placement of nodes within the tree, that describes the data type of the object.

See Also

12.2 Constructors

Synopsis

```
#include <ProbeType.h>
ProbeType(void)
```

Description

The default constructor creates an object with data type of `DEN_default_type`.

See Also

12.3 child

Synopsis

```
#include <ProbeType.h>

ProbeType child(int index) const
```

Parameters

`index`	index of the sub-type, which must be greater than or equal to zero, and less than `child_count()`

Description

This function returns the sub-type of a data type. For example, if the invoking object represents a pointer to an object, `child(0)` returns the data type of the pointee. For data types representing functions, `child(0)` returns the data type of the return value, `child(1)` returns the data type of the first argument, if any, `child(2)` returns the data type of the second argument, if any, *etc*. If the `index` is less than zero or greater than or equal to `child_count()`, a data type of `DEN_error_type` is returned.

Return value

The data type of the indicated sub-type or an undefined data type.

See Also

12.4 child_count

Synopsis

```
#include <ProbeType.h>
int child_count(void) const
```

Description

This function returns the number of sub-types associated with this data type. Undefined data types, created by the default constructor, return zero. Children can be the data type of a pointee, function return types, function argument data types, *etc*.

Return value

Number of child sub-types associated with this data type.

See Also

12.5 function_type

Synopsis

```
#include <ProbeType.h>
friend ProbeType function_type(
     ProbeType return_type,
     int count,
     ProbeType *args)
```

Parameters

`return_type`	data type of the function return value
`count`	number of function arguments
`args`	array of argument data types

Description

This function creates a data type that represents the prototype or type signature of a function.

Return value

Data type that represents the prototype of a function.

See Also

12.6 get_node_type

Synopsis

```
#include <ProbeType.h>
DataExpNodeType get_node_type(void) const
```

Description

This function returns the enumeration value, or node type, of this node in the data type expression tree.

Return value

Node type of this node in the data type expression tree.

See Also

12.7 int32_type

Synopsis

```
#include <ProbeType.h>
friend ProbeType int32_type(void)
```

Description

This function creates an object that represents a 32-bit integer data type.

Return value

Data type that represents a 32-bit integer.

See Also

12.8 operator =

Synopsis

```
#include <ProbeType.h>
ProbeType &operator = (const ProbeType &copy)
```

Parameters

`copy`	probe type to be duplicated

Description

This function transfers the contents of the `copy` parameter to the object.

Return value

Reference to the object.

See Also

12.9 operator ==

Synopsis

```
#include <ProbeType.h>

int operator == (const ProbeType &compare)
```

Parameters

`compare` probe type to be compared

Description

This function compares two probe types for equivalence. If the two data types are equivalent, this function returns 1. Otherwise it returns 0.

Return value

This function returns 1 if the two data types are equivalent, 0 otherwise.

See Also

12.10 operator !=

Synopsis

```
#include <ProbeType.h>

int operator != (const ProbeType &compare)
```

Parameters

`compare`	probe type to be compared

Description

This function compares two probe types for equivalence. If the two data types are equivalent, this function returns 0. Otherwise it returns 1.

Return value

This function returns 0 if the two types are equivalent, 1 otherwise.

See Also

12.11 pointer_type

Synopsis

```
#include <ProbeType.h>
friend ProbeType pointer_type(const ProbeType &pointee)
```

Parameters

`pointee`	data type the pointer will point to

Description

This function creates an object that represents the data type of a pointer to a pointee.

Return value

Data type that represents a pointer to a pointee.

See Also

12.12 stack-LY

Synopsis

```
#include <ProbeType.h>
ProbeExp stack(void *init_val)
```

Parameters

`init_val`	initial value to be given to the stack reference when the reference is allocated on the stack

Description

This function converts a data type into a probe expression that represents a stack reference.

Return value

A probe expression that represents a stack reference.

See Also

12.13 unspecified_type

Synopsis

```
#include <ProbeType.h>
friend ProbeType unspecified_type(int size)
```

Parameters

`size` number of bytes objects of this data type require

Description

This function creates an object that represents an unspecified data type, and has a type value of `DEN_unspecified_type`. The data type must be given a size greater than zero.

Return value

Data type that represents an unspecified data type.

See Also

13.0 class Process

13.1 Supporting Data Types

13.1.1 ConnectState

Synopsis

```
#include <Process.h>
enum ConnectState {
     PRC_connected,
     PRC_attached,
     PRC_created,
     PRC_unknown_state
     PRC_unconnected
     PRC_destroyed
     PRC_pre_create,
     PRC_LAST_CONNECT_STATE
}
```

Description

This enumeration type is used to describe the state of DPCL Processes [need figure showing states]. The state a Process governs the actions that the user can perform.

13.2 Constructors

Synopsis

```
#include <Process.h>
Process(void)
Process(const Process &copy)
Process(const char *host_name, int task_pid, int task_num = 0)
```

Parameters

`copy`	object to be copied into the new Process object
`host_name`	host name or IP address where the process is located. If 0 then the process is considered local

`task_pid` process id for the task

`task_num` task number for the given process

Description

The default constructor creates a Process object in an "unused" state. Specifically, the task number and process ID are both -1, and the host name is 0.

The copy constructor uses the values contained in the `copy` argument to initialize the new (constructed) object. No attempt is made to connect to the process represented by the `copy` argument, whether or not it is already connected.

The standard constructor uses the arguments provided to initialize the object. No attempt is made to connect to the process. `Task_num` is a value that is used only by queries on the client and does not affect the connection in any way.

Exceptions

Exceptions that could be raised as a result of calling this function are unknown at this time.

`AisStatus` ???

See Also

`connect, bconnect, bdisconnect, disconnect, remove_process.`

13.3 activate_probe

Synopsis

```
#include <Process.h>
AisStatus activate_probe(
     short count,
     ProbeHandle *phandle,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`count`	number of probe expressions in the list to be activated
`phandle`	array of probe handles, one for each probe expression to be activated
`ack_cb_fp`	acknowledgement callback function to be invoked when *all* probe expressions in the array have been activated (or activation fails)
`ack_cb_tag`	tag to be used with the acknowledgement callback function

Description

This function activates a list of probes that have been installed within a process. The activation is atomic in the sense that all probes are activated or all probes fail to be activated for the process.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

To activate a set of probes the process must have been previously connected, and the probes must have been previously installed in that process.

Note that the function submits the request to activate the probes and returns immediately. The acknowledgement callback function receives notification of the success or failure of the activation.

Return value

The return value indicates whether the request for activation was successfully submitted, but indicates nothing about whether the request itself was successfully executed.

`ASC_success`	all activations were successfully submitted
`ASC_???`	

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once for each process for which a probe activation is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	probes were successfully activated on this process
`ASC_operation_failed`	attempt to activate these probes in this process failed

See Also

`bactivate_probe, bconnect, bdisconnect, bprobe_deactivate, bprobe_install, class Process, connect, disconnect, GCBFuncType, probe_deactivate, probe_install, ProbeHandle::activate`

13.4 add_phase

Synopsis

```
#include <Process.h>
AisStatus add_phase(
     const Phase &ps,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)

AisStatus add_phase(
     const Phase &ps,
     ProbeExp init_func,
     GCBFuncType init_cb_fp,
     GCBTagType init_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`ps`	data structure local to the client containing the characteristics of the phase to be created
`init_func`	initialization function that is executed once within the application when the phase is installed
`init_cb_fp`	callback function to handle messages from the initialization function
`init_cb_tag`	tag to be used with the initialization callback function
`ack_cb_fp`	acknowledgement callback function to be invoked each time the phase has been created within a process
`ack_cb_tag`	tag to be used with the acknowledgement callback function

Description

This function adds a new phase structure to the process. A process *must* be connected in order to add a new phase. The phase does not execute for the first time until the amount of time indicated by the phase period has elapsed, starting from the time the phase is added to the process.

The return value indicates whether the request for phase addition was successfully submitted, but indicates nothing about whether the request itself was successfully executed.

The initialization function must be loaded into the application before this operation may take place. The function prototype for the initialization function is:

- `void init_func(void *msg_handle)`

Return value

`ASC_success`	phase addition request was successfully submitted
`ASC_???`	

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

`init_cb_fp`. This callback function is invoked each time the corresponding function in the process instrumentation -- `init_func` -- sends a message to the client. The message format is determined by the function that sends the message.

`ack_cb_fp`. This callback function is invoked once for each process for which a phase addition is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	phase was successfully added to this process
`ASC_operation_failed`	attempt to add a phase to this process failed

See Also

`badd_phase, bconnect, bdisconnect, class GenCallBack, class ProbeModule, class Process, connect, disconnect, GCBFuncType, GCBTagType, alloc_mem, free_mem.`

13.5 alloc_mem

Synopsis

```
#include <Process.h>
ProbeExp alloc_mem(
     ProbeType pt,
     void *init_val,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag,
     AisStatus &stat)

ProbeExp alloc_mem(
     ProbeType pt,
     void *init_val,
     const Phase &ps,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag,
     AisStatus &stat)
```

Parameters

`pt`	data type of the allocated data
`init_val`	pointer to the initial value of the allocated data, or 0 if no initial value is desired
`ps`	phase that will contain the allocated data
`ack_cb_fp`	callback function to process the acknowledgement message
`ack_cb_tag`	tag to be used as an argument to the acknowledgement callback when it is invoked
`stat`	output value indicating the completion status of the function

Description

This function allocates a block of probe data in a process. It returns a single probe expression that may be used to reference the allocated data. The data may be referenced in a probe expression that may be installed in the process.

Note that `alloc_mem` returns control to the caller immediately and does not wait until it has either succeeded or failed on the process. The probe expression representing the allocation is returned immediately whether or not allocation succeeds. The returned probe expression may be used as a data reference on the process if the allocation succeeds. If the data reference is used in another probe expression and the client attempts to install that probe expression in a process where the allocation failed, that probe expression will fail to install. Similarly, installation will fail if one attempts to install the probe in a process where the data was not allocated.

`Stat` indicates whether all requests for allocation were successfully submitted. If all requests are successfully submitted `stat` is given the value `ASC_success`. If some request cannot be submitted then `stat` is given the value `ASC_operation_failed`. It reflects the highest severity encountered.

Return value

A probe expression that may be used as a valid reference to the data on this process if the data is allocated

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the acknowledgement message is received, and then removed. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	data was successfully allocated in this process
`ASC_operation_failed`	attempt to allocate data in this process failed

See Also

`free_mem, balloc_mem, bfree_mem`

13.6 attach

Synopsis

```
#include <Process.h>
AisStatus attach(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to be invoked with a successful or failed attachment to this process.
`tag`	callback tag to be used as a parameter to the callback when the callback function is invoked.

Description

Attach to this process. When multiple tools are connected to a process or application, only one tool can be attached at a time. Attaching to a process allows the tool to control the execution directly, such as suspending and resuming execution. Processes must first be connected or created before they can be attached.

Note that the function submits the request to attach to a process and returns immediately. The callback function receives notification of the success or failure of attachment.

Return value

The return value for `attach` indicates whether the request was successfully submitted, but indicates nothing about whether the request itself was successfully executed.

`ASC_success`	request to attach was successfully submitted
`ASC_operation_failed`	attempt to request attachment to the process failed, perhaps because the process is not connected

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once for each process for which an attach is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully attached
`ASC_operation_failed`	attempt to attach to this process failed
ASC_duplicate_attach	already attached

See Also

`battach, bdetach, detach`

13.7 bactivate_probe

Synopsis

```
#include <Process.h>

AisStatus bactivate_probe(short count, ProbeHandle *phandle)
```

Parameters

`count`	number of probe expressions in the list to be activated
`phandle`	array of probe handles, one for each probe expression to be activated

Description

This function activates a list of probes that have been installed within a process. The activation is atomic in the sense that all probes are activated or all probes fail to be activated for any given process.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

To activate a set of probes the process must have been previously connected, and the probes must have been previously installed in the process.

Note that the function submits the request to activate the probes and waits until the request has completed.

Return value

The return value indicates whether the request for activation was successfully executed.

`ASC_success`	all activations were successfully completed
`ASC_operation_failed`	all activations failed

Exceptions

Exceptions that could be raised as a result of calling this function are unknown at this time.

`AisStatus`	???

See Also

`activate_probe, bconnect, bdisconnect, bprobe_deactivate, bprobe_install, connect, disconnect, probe_deactivate, probe_install.`

13.8 badd_phase

Synopsis

```
#include <Process.h>
AisStatus badd_phase(const Phase &ps)
AisStatus badd_phase(
     const Phase &ps,
     ProbeExp init_func,
     GCBFuncType init_cb_fp,
     GCBTagType init_cb_tag)
```

Parameters

`ps`	data structure local to the client containing the characteristics of the phase to be created
`init_func`	initialization function that is executed once within the application when the phase is installed
`init_cb_fp`	callback function to handle messages from the initialization function
`init_cb_tag`	tag to be used with the initialization callback function

Description

This function adds a new phase structure to a connected process. A process *must* be connected in order to add a new phase. The phase does not execute for the first time until the amount of time indicated by the phase period has elapsed, starting from the time the phase is added to the process.

Note that the function submits a request to add the phase and waits until the request has completed. The return value indicates whether the request was successfully executed.

The initialization function must be loaded into the application before this operation may take place. The function prototype for the initialization function is:

- `void init_func(void *msg_handle)`

Return value

The return value indicates whether the request for phase addition was successfully executed.

`ASC_success`	phase was successfully added to the process
`ASC_operation_failed`	phase addition failed

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked each time the corresponding function in the process instrumentation -- `init_func` -- sends a message to the client. The message format is determined by the function that sends the message.

See Also

`add_phase, bconnect, bdisconnect, class ProbeModule, connect, disconnect, alloc_mem, free_mem.`

13.9 balloc_mem

Synopsis

```
#include <Process.h>
ProbeExp balloc_mem(ProbeType pt, void *init_val, AisStatus
&stat)


ProbeExp balloc_mem(
     ProbeType pt,
     void *init_val,
     const Phase &ps,
     AisStatus &stat)
```

Parameters

`pt`	data type of the allocated data
`init_val`	pointer to the initial value of the allocated data, or 0 if no initial value is desired
`ps`	phase that will contain the allocated data
`stat`	output value indicating the completion status of the function

Description

This function allocates a block of probe data in a process. It returns a single probe expression that may be used to reference the allocated data. The data may be referenced in a probe expression that may be installed in the process.

Note that `balloc_mem` does not return control to the caller until it has either succeeded or failed on the process. If the allocation succeeds it returns a valid probe expression data reference and `stat` is given the value `ASC_success`. If the allocation fails then `stat` is given the value `ASC_operation_failed` and any probe that references the returned value of `balloc_mem` will fail to install.

Return value

A probe expression that may be used as a valid reference to the data on this process.

See Also

`bfree_mem, free_mem, alloc_mem`

13.10 battach

Synopsis

```
#include <Process.h>
AisStatus battach(void)
```

Description

Attach to a process. When multiple tools are connected to a process or application, only one tool can be attached at a time. Attaching to a process or application allows the tool to control the execution directly, such as suspending and resuming the process. Processes must first be connected or created before they can be attached.

Note that `battach` does not return control to the caller until the attachment has either succeeded or failed. The return value indicates whether the attachment succeeded or failed.

Return value

The return value for `battach` indicates whether the attachment was successfully established.

`ASC_success`	process was successfully attached as expected.
`ASC_operation_failed`	the process failed to attach
ASC_duplicate_attach	already attached

See Also

`attach, bdetach, detach`

13.11 bconnect

Synopsis

```
#include <Process.h>
AisStatus bconnect(void)
```

Description

Connect to a process. Connection to a process establishes a communication channel to the CPU where the process resides and creates the environment within that process that allows the client to insert and remove instrumentation, *etc*.

Connections from multiple DPCL based tools to the same process are allowed.

Note that `bconnect` does not return control to the caller until the connection has either succeeded or failed. The return value indicates whether the connection succeeded or failed.

Return value

The return value for `bconnect` indicates whether the connection was successfully established.

`ASC_success`	connection was successfully established as expected.
`ASC_operation_failed`	connection failed to be established.

See Also

`bdisconnect, connect, disconnect`

13.12 bcreate

Synopsis

```
#include <Process.h>
AisStatus bcreate(
     const char *host,
     const char *path,
     const char *args[],
     const char *envp[],
     char *remote_stdin_filename,
     char *remote_stdout_filename,
     char *remote_stderr_filename,
     GCBFuncType stdout_cb_fp,
     GCBTagType stdout_cb_tag,
     GCBFuncType stderr_cb_fp,
     GCBTagType stderr_cb_tag)

AisStatus bcreate(
     const char *host,
     const char *path,
     const char *args[],
     const char *envp[],
     GCBFuncType stdout_cb_fp,
     GCBTagType stdout_cb_tag,
     GCBFuncType stderr_cb_fp,
     GCBTagType stderr_cb_tag)

AisStatus bcreate(
     const char *host,
     const char *path,
     const char *args[],
```

```
        const char *envp[],
        char *remote_stdin_filename,
        char *remote_stdout_filename,
        char *remote_stderr_filename)

AisStatus bcreate(
        const char *host,
        const char *path,
        const char *argv[],
        const char *envp[])
```

Parameters

`host`	host name or IP address of the host machine where the process is to be created
`path`	complete path to the executable program, including executable name and relative or absolute directory, when appropriate
`argv`	null terminated array of arguments to be provided to the executable
`envp`	null terminated array of environment variables to be provided to the executable

remote_stdin_filenameremote file to use for stdin

remote_stdout_filenameremote file to use for stdout

remote_stderr_filenameremote file to use for stderr

`stdout_cb_fp`	callback function to handle *stdout* from the process
`stdout_cb_tag`	tag to be used with the *stdout* callback function
`stderr_cb_fp`	callback function to handle *stderr* from the process
`stderr_cb_tag`	tag to be used with the *stderr* callback function

Description

This function creates a process on the specified host. The process is created in a stopped state, and a connection is established that allows the client to insert instrumentation into the created process. The process must be started to begin execution.

The input, output filenames, output callbacks and PoeAppl::send_stdin can be used to access the stdio from and to the process.

If you pass callback functions in to the stdout_cb_fp and stderr_cb_fp parameters, the output from the process will be available in these callbacks. Input to the process can be sent using send_stdin().

Another way to access Stdio to the process is to specify the remote filename parameters. In this case stdin, stdout and stderr can be set to use files on the host where process is running. It is expected that the remote_stdin_filename specified will already exist. The files for the remote_stdin_filename and remote_stdin_filename will created or overwritten if they already exist. If one of the remote file parameters is specified, it takes precedence over the corresponding callback or send_stdin() method of handling Stdio.

Note that `bcreate` does not return control to the caller until the new process has been created or failed to be created. The return value indicates whether the operation succeeded or failed.

Return value

The return value for `bcreate` indicates whether the process was successfully created.

`ASC_success`	process was successfully created, as expected
`ASC_operation_failed`	process failed to be created

Callback Data

`stdout_cb_fp`. This callback function is invoked each time the process sends data to `stdout`.

`stderr_cb_fp`. This callback function is invoked each time the process sends data to `stderr`.

The output will be contained in the message parameter of the callback. The size of the output will be contained in the msg_size field of the sys callback parameter. The output from the process may be recieved in different size blocks than were actually sent by the program.

See Also

`bdestroy, bstart, create, destroy, send_stdin, start`

13.13 bdeactivate_probe

Synopsis

```
#include <Process.h>
AisStatus bdeactivate_probe(short count, ProbeHandle *phandle)
```

Parameters

`count`	number of probes to be deactivated
`phandle`	array of probe handles, representing the probes, to be deactivated

Description

This function accepts an array of probe handles as an input parameter. Each probe handle in the array represents a probe that has been installed in the application. The client sends a request to each of the processes within the application to deactivate the list of probes represented by the array. Probes are deactivated atomically for each process in the sense that the process is temporarily stopped, all probes on the list are deactivated, then the process is resumed. None of the probes in the array are left active.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

Note that `bdeactivate_probe` does not return control to the caller until all probes in the array have been deactivated on the process. The return value indicates whether all probes in the list were deactivated or one or more probes were left intact.

Return value

The return value for `bdeactivate_probe` indicates whether the deactivations were successfully completed.

`ASC_success`	all probe deactivations completed as expected
`ASC_operation_failed`	all probe deactivations failed

See Also

13.14 bdestroy

Synopsis

```
#include <Process.h>

AisStatus bdestroy(void)
```

Description

This function destroys or terminates the processes.

Note that `bdestroy` does not return control to the caller until the process has been destroyed or has failed to be destroyed. The return value indicates whether the termination succeeded or failed.

Return value

The return value for `bdestroy` indicates whether the termination successfully completed.

`ASC_success`	process was successfully terminated, as expected
ASC_no_destroy_from_connected	process must be in attached state to call destory
`ASC_operation_failed`	termination failed

See Also

`destroy`

13.15 bdetach

Synopsis

```
#include <Process.h>
AisStatus bdetach(void)
```

Description

This function detaches the process. Process control flow, such as suspending and resuming the process, can only be done while a process is in an attached state. Detaching a process removes the level of process control available to the client or tool when the process is attached, but retains the process connection so probe installation, activation, removal, *etc*. can still take place.

Note that `bdetach` does not return control to the caller until the process has been detached or failed to do so. The return value indicates whether the process successfully detached or failed to detach.

Return value

The return value for `bdetach` indicates whether the process was successfully detached.

`ASC_success`	process was successfully detached, as expected
ASC_no_detach_from_created	currently created, must attach before detaching
ASC_no_detach_from_connected	currently connected, must attach before detaching
`ASC_operation_failed`	process failed to detach

See Also

`attach, battach, detach`

13.16 bdisconnect

Synopsis

```
#include <Process.h>

AisStatus bdisconnect(void)
```

Description

Disconnect from the process. Disconnecting from an application process removes the application environment created by a connection. All instrumentation and data are removed from the application process.

Note that `bdisconnect` does not return control to the caller until the process has either succeeded or failed in disconnecting.

Return value

The return value for `bdisconnect` indicates whether the connection was successfully terminated.

`ASC_success`	connection was successfully terminated as expected
`ASC_operation_failed`	connection failed to terminate

See Also

`bconnect, connect, disconnect`

13.17 bexecute

Synopsis

```
#include <Process.h>
AisStatus bexecute(
     ProbeExp pexp,
     GCBFuncType data_cb_fp,
     GCBTagType data_cb_tag)
```

Parameters

`pexp`	probe expression to be executed in the application process
`data_cb_fp`	callback function to be invoked when data from the probe is received
`data_cb_tag`	callback tag to be used when the data callback function is invoked

Description

This function executes a probe expression within the application process. The expression is executed once, then removed. The application process is interrupted, the expression is executed, then the process resumes execution as before the interruption.

Note that `bexecute` does not return control to the caller until the probe expression has either succeeded or failed to execute.

Return value

The return value for `bexecute` indicates whether the request for execution succeeded or failed.

`ASC_success`	probe expression was successfully executed
`ASC_operation_failed`	attempt to execute the probe expression failed

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once for each message sent from the probe. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback tag is given in the `data_cb_tag` variable. The callback message is the data send by the probe using the `Ais_send()` function call.

See Also

`execute, Ais_send`

13.18 bfree_mem

Synopsis

```
#include <Process.h>

AisStatus bfree_mem(ProbeExp pexp)
```

Parameters

`pexp`	dynamically allocated block of probe memory

Description

This function deallocates a block of dynamically allocated probe memory in an application process. The probe expression must contain only a single reference to a block of data allocated by the `alloc_mem` or `balloc_mem` functions.

Note that `bfree` does not return control to the caller until deallocating the block of memory has either succeeded or failed.

Return value

The return value for `bfree_mem` indicates whether the requests for deallocation were successfully executed.

See Also

`free_mem, bfree_mem, balloc_mem`

13.19 binstall_probe

Synopsis

```
#include <Process.h>
AisStatus binstall_probe(
     short count,
     ProbeExp *probe_exp,
     InstPoint *point,
     GCBFuncType *data_cb_fp,
     GCBTagType *data_cb_tag,
     ProbeHandle *phandle)
```

Parameters

`count`	number of probe expressions to be installed
`probe_exp`	probe expressions to be installed
`point`	instrumentation points where the probe expressions are to be installed
`data_cb_fp`	callback functions to process data received from the probe expression
`data_cb_tag`	tags to be used as an argument to the data callback when it is invoked
`phandle`	probe handles that represent the installed probe expressions

Description

This function installs probe expressions as instrumentation at specific locations within the process. Probe expressions are installed atomically, in the sense that within a process either all probe expressions in the request are installed into the process, or none of the expressions are installed. The return value indicates whether all probes were installed, or whether the process was unable to install the expressions as requested.

`Data_cb_fp` is an input array supplied by the caller that must contain at least `count` elements. The i^{th} element of the array is a pointer to a callback function that is invoked each time the i^{th} probe in `phandle` sends data via the `AisSendMsg` function. `Data_cb_tag` is a similar array that contains the callback tag used when callbacks in `data_cb_fp` are invoked. The i^{th} callback tag is used with the i^{th} callback.

`Phandle` is an output array supplied by the caller that must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, to be used in subsequent references to the i^{th} probe expression. For example, it is needed when the client activates, deactivates or removes a probe expression from an application or process. `Phandle` does not contain valid information if the installation fails.

Note that `binstall_probe` does not return control to the caller until all probe expressions have been installed or failed to install within the process.

Return value

The return value for `binstall_probe` indicates whether the probe installations were successful.

`ASC_success`	all probes were successfully installed, as expected
`ASC_operation_failed`	one or more of the probes could not be installed as requested, so none of the probes were installed

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once for each message sent from the probe. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback tag is given in the `data_cb_tag` array. The callback message is the data send by the probe using the `Ais_send` function call.

See Also

`Ais_send, install_probe, ...`

13.20 bload_module

Synopsis

```
#include <Process.h>

AisStatus bload_module(ProbeModule *module)
```

Parameters

module	the probe module to be loaded.

Description

This function sends and loads the module from the client side to the user application. Once loaded, the probe expressions available in this probe module can be installed and activated as if those are native in the application.

Note that `bload_module` does not return control to the caller until the probe module has been installed or failed to install in the process.

Return value

The return value for `bload_module` indicates whether the probe module installation was successful.

`ASC_success`	module was successfully installed on all processes
`ASC_operation_failed`	module could not be installed as requested on one or more processes

See Also

`bunload_module, load_module, unload_module`

13.21 breadmem - LY

Synopsis

```
#include <Process.h>
AisStatus breadmem(char *location, char *buffer, int size)
```

Parameters

`location`	address in the application process where reading is to begin
`buffer`	address in the client process where data is to be placed
`size`	size, in bytes, of both the buffer and the memory block to be read

Description

This function sends a request to the daemon managing this process to read the indicated block of memory within the process. The block of memory is then returned to the client and stored in the indicated buffer.

Note that `breadmem` does not return control to the caller until the memory has been read or failed to be read from the process.

Return value

The return value for `breadmem` indicates whether the block of memory was successfully read from the application process.

`ASC_success`	memory was successfully read, as expected
`ASC_operation_failed`	memory could not be read

See Also

`bwritemem, readmem, writemem`

13.22 bremove_phase

Synopsis

```
#include <Process.h>
AisStatus bremove_phase(const Phase &ps)
```

Parameters

`ps`	phase description to be removed from the application

Description

This function removes a phase from the application. Data and functions associated with the phase are unaffected by removing the phase. Existing probe data cannot become associated with a phase except at the time of data allocation, so deleting a phase has the effect of permanently disassociating data from any phase.

Note that `bremove_phase` does not return control to the caller until the phase has been removed or failed to be removed from the process.

Return value

The return value for `bremove_phase` indicates whether the phase was successfully removed from the process.

`ASC_success`	phase was successfully removed, as expected
`ASC_operation_failed`	phase could not be removed from the process

See Also

`remove_phase, add_phase, badd_phase, set_phase_exit, bset_phase_exit, set_phase_period, bset_phase_period, get_phase_period`

13.23 bremove_probe

```
#include <Process.h>
```

```
AisStatus bremove_probe(short count, ProbeHandle *phandle)
```

Parameters

`count`	number of probe handles in the accompanying array
`phandle`	array of probe handles representing probe expressions to be removed

Description

This function deletes or removes probe expressions that have been installed in a process. If all probe expressions are installed and deactivated, the probe expressions are removed and a "normal" return status results. If one or more of the probe expressions are currently active, the expressions are deactivated and removed, and the return status indicates there were active probes at the time of their removal. If one or more of the probes do not exist, all existing probes are removed and the return status indicates an appropriate warning. If one or more of the probe expressions exists but cannot be removed, an error results and none of the probe expressions is removed. If the process is not connected a warning is returned.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

Probe expression removal is atomic in the sense that all probe expressions are removed from a given process or none are. When probes are removed from a process the process is temporarily stopped, all indicated probes are removed, and the process is resumed.

Note that `bremove_probe` does not return control to the caller until the probes have been removed or failed to be removed from the process. If one or more probes cannot be removed for any reason, as many as can are removed and status indicates the condition.

Return value

The return value for `bremove_probe` indicates whether all probes in the list were successfully removed from the process.

`ASC_success`	all probes were successfully removed, as expected
`ASC_operation_failed`	one or more of the probes were not removed

See Also

`bactivate_probe, bdeactivate_probe, binstall_probe, activate_probe, deactivate_probe, install_probe, remove_probe`

13.24 bresume

Synopsis

```
#include <Process.h>
AisStatus bresume(void)
```

Description

This function resumes execution of a process that has been temporarily suspended by a suspend or `bsuspend` function call. A process must be attached for it to be resumed. A process that is not attached will result in a error return code.

Note that `bresume` does not return control to the caller until the process has resumed or failed to resume.

Return value

The return value for `bresume` indicates whether the process was successfully resumed.

`ASC_success`	process was resumed, as expected
`ASC_operation_failed`	process failed to be resumed
ASC_no_sus_res_from_created	must be attached to call bresume
ASC_no_sus_res_from_connected	must be attached to call bresume

See Also

`attach, battach, bconnect, bdetach, bdisconnect, bsuspend, connect, detach, disconnect, resume, suspend`

13.25 bset_phase_exit

Synopsis

```
#include <Process.h>
AisStatus bset_phase_exit(
     const Phase &ps,
     ProbeExp begin_func,
     GCBFuncType begin_cb_fp,
     GCBTagType begin_cb_tag,
     ProbeExp iter_func,
     GCBFuncType iter_cb_fp,
     GCBTagType iter_cb_tag,
     ProbeExp end_func,
     GCBFuncType end_cb_fp,
     GCBTagType end_cb_tag)
```

Parameters

`ps`	phase description to be removed from the application
`begin_func`	initialization function that is executed once within the application when the phase is removed
`begin_cb_fp`	callback function to handle messages from the initialization function
`begin_cb_tag`	tag to be used with the initialization callback function
`iter_func`	iteration function that is executed within the application on each piece of data associated with the phase when the phase is removed
`iter_cb_fp`	callback function to handle messages from the iteration function
`iter_cb_tag`	tag to be used with the iteration callback function
`end_func`	termination function that is executed once within the application when the phase is removed
`end_cb_fp`	callback function to handle messages from the termination function
`end_cb_tag`	tag to be used with the termination callback function

Description

This function specifies a set of exit functions to be executed when any of the following three events occur.

- when the indicated phase is removed using either the remove_phase or bremove_phase function call
- when diconnecting from the target process (without calling remove_phase or bremove_phase first)
- when the target process has finished execution while the indicated phase is still active

Note that `set_phase_exit` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the exit functions have been placed in the indicated phase or the operation failed to complete.

Each of the phase functions must be loaded into the application before this operation may take place. The function prototypes for the functions are:

- `void begin_func(void *msg_handle)`
- `void iter_func(void *msg_handle, void *data)`
- `void end_func(void *msg_handle)`

Return value

The return value for `set_phase_exit` indicates whether the request to set exit functions for the indicated phase on the process was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	remove request was successfully submitted
`ASC_operation_failed`	remove operation failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

begin_cb_fp, **iter_cb_fp**, **end_cb_fp**. These callback functions are invoked each time the corresponding function in the process instrumentation -- `begin_func`, `iter_func`, or `end_func` -- sends a message to the client. The message format is determined by the function that sends the message.

See Also

`set_phase_exit, add_phase, badd_phase, remove_phase, bremove_phase, set_phase_period, bset_phase_period, get_phase_period`

13.26 bset_phase_period

Synopsis

```
#include <Process.h>
AisStatus bset_phase_period(const Phase &ps, float period)
```

Parameters

`ps`	phase to be modified
`period`	new time interval between successive phase activations, in seconds

Description

This function changes the time interval between successive activations of a phase within the process. Processes which do not have the phase installed result in an informational return code. Processes that are not connected result in a warning return code.

The new period is represented by a floating-point value. If the value is positive it represents the time interval in seconds. If the value is zero or positive and smaller than the minimum activation time interval, it represents the minimum activation delay time. In both cases the phase is activated immediately before setting the new interval. If the value is less than zero the phase is disabled immediately, but left in place for possible future reactivation.

Note that `bset_phase_period` does not return control to the caller until the phase period has been set or failed to be set in the process.

Return value

The return value for `bset_phase_period` indicates whether the phase period was successfully set on this process.

`ASC_success`	phase period was successfully set
`ASC_operation_failed`	phase period failed to be set

See Also

`add_phase, badd_phase, bremove_phase, get_phase_period, remove_phase, set_phase_period`

13.27 bsignal - LY

Synopsis

```
#include <Process.h>
AisStatus bsignal(int unix_signal)
```

Parameters

`unix_signal`	Unix™ signal to be sent to every process in the application

Description

This function sends the specified signal to the process. The process must be both connected and attached to receive the signal. The function does not return until the process receives and acknowledges receiving the signal.

A signal is sent only to those processes that are connected and attached.

Note that `bsignal` does not return control to the caller until the process has been signalled or failed to be signalled.

Return value

The return value for `bsignal` indicates whether the AIX signal was successfully sent to the process.

`ASC_success`	signal was successfully sent to the process
`ASC_operation_failed`	signal failed to be sent to the process

See Also

`signal`

13.28 bstart

Synopsis

```
#include <Process.h>
AisStatus bstart(void)
```

Description

This function starts the execution of a process that has been created but not yet begun execution. When applied to a process that has begun execution it causes the process to terminate and restart.

Note that `bstart` does not return control to the caller until the process has started or failed to start.

Return value

The return value for `bstart` indicates whether the process was successfully started.

`ASC_success`	process was started
`ASC_operation_failed`	process failed to be started

See Also

`bcreate, bdestroy, create, destroy, start`

13.29 bsuspend

Synopsis

```
#include <Process.h>
AisStatus bsuspend(void)
```

Description

This function suspends a process that is executing. A tool must be attached to a process in order to suspend process execution.

Note that `bsuspend` does not return control to the caller until the process has been suspended or failed to be suspended.

Return value

The return value for `bsuspend` indicates whether all processes within the application were successfully suspended.

`ASC_success`	process was successfully suspended
`ASC_operation_failed`	process failed to be suspended
ASC_no_sus_res_from_created	must be attached to be suspended
ASC_no_sus_res_from_connected	must be attached to be suspended

See Also

`battach, bresume, attach, resume, suspend`

13.30 bunload_module

Synopsis

```
#include <Process.h>
AisStatus bunload_module(ProbeModule* module)
```

Parameters

`module`	probe module to be removed from the application process

Description

This function unloads the module from process. Once unloaded, All the probe handles that refer to this probe module are automatically removed.

Note that `bunload_module` does not return control to the caller until the probe module has been removed or failed to be removed from the application process.

Return value

The return value for `bunload_module` indicates whether the probe module was successfully removed from the process.

`ASC_success`	module was successfully removed from the process
`ASC_operation_failed`	module could not be removed from the process

See Also

`bload_module, load_module, unload_module`

13.31 bwritemem -LY

Synopsis

```
AisStatus bwritemem(char *location, char *buffer, int size)
```

Parameters

`location`	address in the application process where writing is to begin
`buffer`	address in the client process from which data is to be taken
`size`	size, in bytes, of both the buffer and the memory block to be written

Description

This function sends a request to the daemon managing this process to write the indicated block of memory within the process. Data to write the block of memory is taken from the indicated client buffer.

Note that `bwritemem` does not return control to the caller until the memory has been written or failed to be written on the process.

Return value

The return value for `bwritemem` indicates whether the block of memory was successfully written to the application process.

`ASC_success`	memory was successfully written, as expected
`ASC_operation_failed`	memory could not be written

See Also

`breadmem, readmem, writemem`

13.32 connect

Synopsis

```
#include <Process.h>

AisStatus connect(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to be invoked with each successful or failed connection to a process listed within the application
`tag`	callback tag to be used each time the callback function is invoked

Description

Connection to a process establishes a communication channel to the CPU where the process resides (the host CPU) and creates the environment within that process that allows the client to insert and remove instrumentation, alter its control flow, *etc*.

Connections from multiple DPCL based tools to the same process are allowed.

Note that the function submits the requests to connect the process and returns immediately. The callback function receives notification of a connection's success or failure.

Return value

The return value for `connect` indicates whether the request for connection was successfully submitted, but indicates nothing about whether the request was successfully executed.

`ASC_success`	connection request was successfully submitted
`ASC_operation_failed`	request could not be submitted

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once for each process for which a connection is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	connection was successfully established on this process
`ASC_operation_failed`	attempt to connect to this process failed

See Also

`bconnect, bdisconnect, disconnect`

13.33 create

Synopsis

```
#include <Process.h>
AisStatus create(
     const char *host,
     const char *path,
     const char *args[],
     const char *envp[],
     char *remote_stdin_filename,
     char *remote_stdout_filename,
     char *remote_stderr_filename,
     GCBFuncType stdout_cb_fp,
     GCBTagType stdout_cb_tag,
     GCBFuncType stderr_cb_fp,
     GCBTagType stderr_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
AisStatus create(
     const char *host,
     const char *path,
     const char *args[],
     const char *envp[],
     GCBFuncType stdout_cb_fp,
     GCBTagType stdout_cb_tag,
     GCBFuncType stderr_cb_fp,
     GCBTagType stderr_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
AisStatus create(
     const char *host,
```

```
        const char *path,
        const char *args[],
        const char *envp[],
        char *remote_stdin_filename,
        char *remote_stdout_filename,
        char *remote_stderr_filename,
        GCBFuncType ack_cb_fp,
        GCBTagType ack_cb_tag)
    AisStatus create(
        const char *host,
        const char *path,
        const char *args[],
        const char *envp[],
        GCBFuncType ack_cb_fp,
        GCBTagType ack_cb_tag)
```

Parameters

`host`	host name or IP address of the host machine where the process is to be created
`path`	complete path to the executable program, including file name and relative or absolute directory, when appropriate
`args`	null terminated array of arguments to be provided to the executable
`envp`	null terminated array of environment variables to be provided to the executable

remote_stdin_filenameremote file to use for stdin

remote_stdout_filenameremote file to use for stdout

remote_stderr_filenameremote file to use for stderr

`stdout_cb_fp`	callback function to handle *stdout* from the process
`stdout_cb_tag`	tag to be used with the *stdout* callback function
`stderr_cb_fp`	callback function to handle *stderr* from the process
`stderr_cb_tag`	tag to be used with the *stderr* callback function
`ack_cb_fp`	callback function to be invoked with a successful or failed creation
`ack_cb_tag`	callback tag to be used when the callback function is invoked

Description

This function creates a process on the specified host. The process is created in a stopped state, and a connection is established that allows the client to insert instrumentation into the created process. The process must be started to begin execution.

The input, output filenames, output callbacks and PoeAppl::send_stdin can be used to access the stdio from and to the process.

If you pass callback functions in to the stdout_cb_fp and stderr_cb_fp parameters, the output from the process will be available in these callbacks. Input to the process can be sent using send_stdin().

Another way to access Stdio to the process is to specify the remote filename parameters. In this case stdin, stdout and stderr can be set to use files on the host where process is running. It is expected that the remote_stdin_filename specified will already exist. The files for the remote_stdin_filename and remote_stdin_filename will created or overwritten if they already exist. If one of the remote file parameters is specified, it takes precedence over the corresponding callback or send_stdin() method of handling Stdio.

Note that `create` returns control immediately to the caller. It does not wait until the process has been created. The return value indicates whether the request was successfully submitted and gives no indication whatever about the success or failure of the execution of the request.

Return value

The return value for `create` indicates whether the request for process creation was successfully submitted, but indicates nothing about whether the request was successfully executed.

`ASC_success`	process creation request was successfully submitted
`ASC_operation_failed`	request could not be submitted

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

`ack_cb_fp`. This callback function is invoked once when the new process is created. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	connection was successfully established on this process
`ASC_operation_failed`	attempt to connect to this process failed

`stdout_cb_fp`. This callback function is invoked each time the process sends data to `stdout`.

stderr_cb_fp. This callback function is invoked each time the process sends data to stderr.

The output will be contained in the message parameter of the callback. The size of the output will be contained in the msg_size field of the sys callback parameter. The output from the process may be recieved in different size blocks than were actually sent by the program.

See Also

bcreate, bdestroy, bstart, destroy, start

13.34 deactivate_probe

Synopsis

```
#include <Process.h>
AisStatus deactivate_probe(
     short count,
     ProbeHandle *phandle,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`count`	number of probes to be deactivated
`phandle`	array of probe handles, representing the probes, to be deactivated
`ack_cb_fp`	acknowledgement callback function to be invoked when *all* probe expressions in the array have been deactivated (or deactivation fails)
`ack_cb_tag`	tag to be used with the acknowledgement callback function

Description

This function accepts an array of probe handles as an input parameter. Each probe handle in the array represents a probe that has been installed in the application. The client sends a request to each of the processes within the application to deactivate the list of probes represented by the array. Probes are deactivated atomically for each process in the sense that the process is temporarily stopped, all probes on the list are deactivated, then the process is restarted. None of the probes in the array are left active. If one or more probes cannot be deactivated, for whatever reason, all that can be deactivated are deactivated.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

Note that `deactivate_probe` returns control immediately to the caller. It does not wait until all probes in the array have been deactivated on all processes in the application. The return value indicates whether the request was successfully submitted and gives no indication whatever about the success or failure of the execution of the request.

Return value

The return value for `deactivate_probe` indicates whether the deactivations were successfully submitted.

`ASC_success`	all probe deactivations were submitted, as expected
`ASC_operation_failed`	one or more of the probe deactivations were not submitted

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once for each process for which a probe deactivation is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	probes were successfully deactivated on this process
`ASC_operation_failed`	attempt to deactivate probes on this process failed

See Also

13.35 destroy

Synopsis

```
#include <Process.h>
AisStatus destroy(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	acknowledgement callback function to be invoked for each process that is destroyed (or not destroyed)
`tag`	tag to be used with the acknowledgement callback function

Description

This function destroys or terminates all processes within the application.

Note that `destroy` returns control to the caller immediately. It does not wait until all processes within the application have been destroyed. The return value indicates whether the requests were successfully submitted, but give not indication of whether the requests themselves were successfully executed.

Return value

The return value for `destroy` indicates whether the terminations were successfully requested.

`ASC_success`	all terminations were successfully requested, as expected
`ASC_operation_failed`	one or more of the terminations were not requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once when the process destruction is attempted. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully destroyed
ASC_no_destroy_from_connected	process must be in attached state to call destory
`ASC_operation_failed`	attempt to destroy this process failed

See Also

`bdestroy`

13.36 detach

Synopsis

```
#include <Process.h>
AisStatus detach(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to be invoked when detaching from a process succeeds or fails.
`tag`	callback tag to be used when the callback function is invoked.

Description

This function detaches the client from this process. Process control flow, such as suspending and resuming a process, can only be done while a process is in an attached state. Detaching a process removes the level of process control available to the client or tool when the process is attached, but retains the process connection so probe installation, activation, removal, *etc*. can still take place.

Note that `detach` returns control to the caller immediately upon issuing a request to detach from a process. The return value indicates whether the request was successfully submitted.

Return value

The return value for `detach` indicates whether the request was successfully submitted.

`ASC_success`	detach request was successfully submitted, as expected
`ASC_operation_failed`	request was not submitted

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once for each process for which detachment is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully detached
ASC_no_detach_from_created	currently created, must attach before detaching
ASC_no_detach_from_connected	currently connected, must attach before detaching
`ASC_operation_failed`	attempt to detach this process failed

See Also

`attach, battach, bdetach`

13.37 disconnect

Synopsis

```
#include <Process.h>
AisStatus disconnect(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to be invoked when disconnection from a process succeeds or fails.
`tag`	callback tag to be used when the callback function is invoked.

Description

Disconnecting from an application process removes the application environment created by a connection. All instrumentation and data are removed from the application process.

Note that the function submits the request to disconnect the process and returns immediately. The callback function receives notification of a disconnection's success or failure.

Return value

The return value for `disconnect` indicates whether the request for disconnection was successfully submitted, but indicates nothing about whether the request was successfully executed.

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once when the process is (or fails to be) disconnected. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully disconnected
`ASC_operation_failed`	attempt to disconnect this process failed

See Also

`bconnect, bdisconnect, connect`

13.38 execute

Synopsis

```
#include <Process.h>
AisStatus execute(
     ProbeExp probe_exp,
     GCBFuncType data_cb_fp,
     GCBTagType data_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`probe_exp`	probe expression to be executed in the application process
`data_cb_fp`	callback function to be invoked when data from the probe is received
`data_cb_tag`	callback tag to be used when the data callback function is invoked
`ack_cb_fp`	callback function to be invoked when execution succeeds or fails
`ack_cb_tag`	callback tag to be used when the callback function is invoked

Description

This function executes a probe expression within the application process. The expression is executed once, then removed. The application process is interrupted, the expression is executed, then the process resumes execution as before the interruption.

Note that `execute` returns control to the caller immediately upon submitting its request to the daemon. It does not wait until the probe expression has been executed or failed to execute. The acknowledgement callback function receives notification of the success or failure of the execution.

Return value

The return value for `execute` indicates whether the request for deallocation was successfully submitted, but indicates nothing about whether the request was successfully executed.

`ASC_success`	probe expression execution was successfully submitted
`ASC_???`	

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

data_cb_fp. This callback function is invoked once for each message sent from the probe. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback tag is given in the `data_cb_tag` array. The callback message is the data send by the probe using the `Ais_send()` function call.

`ack_cb_fp`. This callback function is invoked once when execution succeeds or fails. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	probe expression was successfully executed
`ASC_operation_failed`	attempt to execute the probe expression failed

See Also

`bexecute, Ais_send`

13.39 free_mem

Synopsis

```
#include <Process.h>
AisStatus free_mem(
     ProbeExp pexp,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`pexp`	dynamically allocated block of probe memory
`ack_cb_fp`	callback function to be invoked when deallocating the block of memory succeeds or fails
`ack_cb_tag`	callback tag to be used when the callback function is invoked

Description

This function deallocates a block of dynamically allocated probe memory for this process. The probe expression must contain only a single reference to a block of data allocated by the `alloc_mem` or `balloc_mem` functions.

Note that `free_mem`

returns control to the caller immediately upon submitting its request to free the data. It does not wait until the data has been deallocated or failed to deallocate. The acknowledgement callback function receives notification of the success or failure of the deallocation.

Return value

The return value for `free_mem` indicates whether the request for deallocation was successfully submitted, but indicates nothing about whether the request was successfully executed.

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once when deallocation succeeds or fails. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	block of probe memory was successfully deallocated
`ASC_operation_failed`	attempt to deallocate memory on this process failed

See Also

`bfree_mem, balloc_mem, alloc_mem`

13.40 get_host_name

Synopsis

```
#include <Process.h>

char *get_host_name(char *buffer, unsigned int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the host name
`len`	maximum number of bytes the function will place in *buffer*. The *len* parameter should include enough space for a terminating *null* byte.

Description

This function copies into *buffer* a *null*-terminated string representing the name of the host machine for the indicated process. The name may be truncated if the *len* parameter is smaller than the length of the host name

Return value

A pointer to *buffer*, which will contain at most *len* bytes of the AIX host machine name.

See Also

`get_host_name_length`

13.41 get_host_name_length

Synopsis

```
#include <Process.h>

unsigned int get_host_name_length(void) const
```

Description

This function returns the length, including the terminating null byte, of the name of the host machine for the indicated process. If there is no host name associated with the process, then a value of zero is returned.

Return value

The length of the the AIX host machine name.

See Also

`get_host_name`

13.42 get_pid

Synopsis

```
#include <Process.h>
int get_pid(void) const
```

Description

This function returns the AIX process identification number for the indicated process.

Return value

AIX process ID.

See Also

13.43 get_phase_period

Synopsis

```
#include <Process.h>

float get_phase_period(const Phase &ps, AisStatus &stat) const
```

Parameters

`ps`	phase being queried on this process
`stat`	output variable that indicates the success or failure of the call

Description

This function returns the time duration, in seconds, between successive activations of this phase. If the return value is greater than zero, the value represents the minimum time between successive activations of the phase. Due to scheduling conflicts with other processes and resources on the system the actual time between phase activations may be greater than the stated value. If the return value is zero it represents the fastest rate of phase activation possible. If the return value is less than zero, it indicates an error.

`Stat` indicates whether the query was successful. To be successful the process must be connected and the phase must exist on the process.

Return value

Minimum time duration, in seconds, between successive activations of this phase.

See Also

13.44 get_program_object

Synopsis

```
#include <Process.h>

SourceObj get_program_object(void) const
```

Description

This function retrieves the top-level source object from the process. Source objects are a coarse source-level view of the program structure. Program objects represent the top level of a tree structure. Below a program object are modules, then data and functions, *etc*. If the process is not connected or some other error occurs, the source object returned will be invalid. The source object may be queried to determine its validity.

Return value

Program object for this process.

See Also

```
class SourceObj
```


13.45 get_task

Synopsis

```
#include <Process.h>
int get_task(void) const
```

Description

This function returns the task identifier associated with this process.

Return value

Task ID for this process.

13.46 install_probe

Synopsis

```
#include <Process.h>
AisStatus install_probe(
     short count,
     ProbeExp *probe_exp,
     InstPoint *point,
     GCBFuncType *data_cb_fp,
     GCBTagType *data_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag,
     ProbeHandle *phandle)
```

Parameters

`count`	number of probe expressions to be installed, instrumentation points, data callback functions, data callback tags, and probe handles
`probe_exp`	probe expressions to be installed
`point`	instrumentation points where the probe expressions are to be installed
`data_cb_fp`	array of callback functions to process data received from the probe expression
`data_cb_tag`	array of tags to be used as an argument to the data callback when it is invoked
`ack_cb_fp`	callback function to process data received from the probe expression
`ack_cb_tag`	tag to be used as an argument to the data callback when it is invoked
`phandle`	probe handles that represent the installed probe expressions

Description

This function installs probe expressions as instrumentation at specific locations within a process. Probe expressions are installed atomically, in the sense that within each process either all probe expressions in the request are installed into the process, or none of the expressions are installed. The return value indicates whether the request to have probes installed was successfully submitted.

`Phandle` is an output array supplied by the caller that must contain at least `count` elements. The ith element of the array is a handle, or identifier, to be used in subsequent references to the ith probe expression. For example, it is needed when the client activates, deactivates or removes a probe expression from an application or process. `Phandle` does not contain valid information if the installation fails.

Note that `install_probe` returns control to the caller immediately upon submitting all requests to the daemons. It does not wait until all probe expressions have been installed or failed to install within all processes within the application.

Return value

The return value for `install_probe` indicates whether the request for probes to be installed was successfully submitted. It gives no indication of whether the requests was successfully executed.

`ASC_success`	probe expression installation request was successfully submitted
`ASC_operation_failed`	probe expression installations failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

ack_cb_fp. The callback function is invoked once and removed. It is called when the status message for this request is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	all probes were successfully installed in this process
`ASC_operation_failed`	attempt to install probes in this process failed

data_cb_fp. The callback function is invoked once for each message sent from the probe. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback tag is given in the `data_cb_tag` array. The callback message is the data send by the probe using the `Ais_send()` function call.

See Also

`activate_probe, bactivate_probe, bdeactivate_probe, bremove_probe, deactivate_probe, remove_probe`

13.47 load_module

Synopsis

```
#include <Process.h>
AisStatus load_module(
     ProbeModule *module,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

module	probe module to be loaded
ack_cb_fp	callback functin to process load module acknowledgements.
ack_cb_tag	tag to be used as an argument to the callback when it is invoked

Description

This function sends and loads the module from the client side to the process. Once loaded, the probe expressions available in this probe module can be installed and activated as if those are native in the application.

Note that `load_module` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the module has been loaded or failed to load within the process.

Return value

The return value for `load_module` indicates whether the request to load the indicated module was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	load requests was successfully submitted
`ASC_operation_failed`	load operation failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once for the process for which disconnection is requested. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	objects were successfully loaded into this process
`ASC_operation_failed`	attempt to load objects on this process failed

See Also

13.48 operator =

Synopsis

```
#include <Process.h>

Process &operator = (const Process &rhs)
```

Parameters

`rhs` right operand

Description

This function assigns the value of the right operand to the invoking object. The left operand is the invoking object. For example, "`Process rhs, lhs; ... lhs = rhs;`" assigns the value of `rhs` to `lhs`. Both values would then refer to the same process, if any.

Note that "`Process x("host", 123), y("host", 123);`" creates two separate "process" data objects, or logical processes, that manipulate the same physical process, but the data objects are managed separately. Thus "`x.connect();`" does not cause the logical process "`y`" to be connected.

Return value

A reference to the invoking object (i.e., the left operand).

See Also

13.49 query_state

Synopsis

```
#include <Process.h>

AisStatus query_state(ConnectState *state)
```

Parameters

`state`	state of the Process object

Description

This function returns the state of the Process. See Figure ? [need figure]. If this state can be determined locally, the function returns immediately. Otherwise, a blocking request is sent to retrieve the state information.

Return value

If the state information cannot be determined locally, the return value for `query_state` indicates whether the request was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	query request was successfully submitted
`ASC_operation_failed`	query operation failed to be requested

See Also

13.50 readmem - LY

Synopsis

```
#include <Process.h>
AisStatus readmem(
     char *location,
     char *buffer,
     int size,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`location`	address in the application process where reading is to begin
`buffer`	address in the client process where data is to be placed
`size`	size, in bytes, of both the buffer and the memory block to be read
`ack_cb_fp`	callback function to process data read from the process
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function sends a request to the daemon managing this process to read the indicated block of memory within the process. The block of memory is then returned to the client in the indicated buffer.

Note that `readmem` returns control to the caller immediately. It does not wait until the memory has been read or failed to be read from the process.

Return value

The return value for `readmem` indicates whether the request to read the block of memory was successfully submitted. It gives no indication whether the request was successfully executed.

`ASC_success`	request was successfully submitted, as expected
`ASC_operation_failed`	request could not be submitted

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the data is received. The data is written to the buffer indicated in the `readmem` function call. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	memory was successfully read in this process
`ASC_operation_failed`	attempt to read memory in this process failed

See Also

`bwritemem, readmem, writemem`

13.51 remove_phase

Synopsis

```
#include <Process.h>
AisStatus remove_phase(
    const Phase &ps,
    GCBFuncType ack_cb_fp,
    GCBTagType ack_cb_tag)
```

Parameters

`ps`	phase description to be removed from the application
`ack_cb_fp`	callback function to process phase removal acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function removes a phase from the application. Data and functions associated with the phase are unaffected by removing the phase. Existing probe data cannot become associated with a phase except at the time of data allocation, so deleting a phase has the effect of permanently disassociating data from any phase.

Note that `remove_phase` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the phase has been removed or failed to be removed from the process.

Return value

The return value for `remove_phase` indicates whether the request to remove the indicated phase on the process was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	remove request was successfully submitted
`ASC_operation_failed`	remove operation failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

ack_cb_fp. The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	phase was successfully removed from this process

`ASC_operation_failed` attempt to remove phase from this process failed

See Also

```
bremove_phase, add_phase, badd_phase, set_phase_exit,
bset_phase_exit, set_phase_period, bset_phase_period,
get_phase_period
```


13.52 remove_probe

Synopsis

```
#include <Process.h>
AisStatus remove_probe(
     short count,
     ProbeHandle *phandle,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`count`	number of probe handles in the accompanying array
`phandle`	array of probe handles representing probe expressions to be removed
`ack_cb_fp`	callback function to process probe removal acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function deletes or removes probe expressions that have been installed in an application. If all probe expressions are installed and deactivated, the probe expressions are removed and a “normal” return status results. If one or more of the probe expressions are currently active, the expressions are deactivated and removed and the return status indicates there were active probes at the time of their removal. If one or more of the probes do not exist, all existing probes are removed and the return status indicates an appropriate warning. If one or more of the probe expressions exists but cannot be removed, an error results and none of the probe expressions is removed. If one or more processes are not connected, probe removal takes place within those that are connected, and a warning is issued.

`Phandle` is an input array generated by an `install_probe` or `binstall_probe` call. It is supplied by the caller and must contain at least `count` elements. The i^{th} element of the array is a handle, or identifier, that identifies the i^{th} probe expression.

Probe expression removal is atomic in the sense that all probe expressions are removed from a given process or none are. When probes are removed from a process the process is temporarily stopped, all indicated probes are removed, and the process is resumed.

Note that `remove_probe` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the probes have been removed or failed to be removed from the process.

Return value

The return value for `remove_probe` indicates whether the request to remove the indicated probes on the process was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	all remove requests were successfully submitted
`ASC_operation_failed`	remove operation failed to be requested to some process

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	probes were successfully removed from this process
`ASC_operation_failed`	attempt to remove probes from this process failed

See Also

`activate_probe, bactivate_probe, bdeactivate_probe, binstall_probe, bremove_probe, deactivate_probe, install_probe`

13.53 resume

Synopsis

```
#include <Process.h>
AisStatus resume(GCBFuncType ack_cb_fp, GCBTagType ack_cb_tag)
```

Parameters

`ack_cb_fp`	callback function to process process resumption acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function resumes execution of an application that has been temporarily suspended by a `stop` or `bstop` function. Execution resumption occurs on a process by process basis. A process must be connected, attached and stopped for it to be resumed. A process that is not connected or not attached will result in a warning return code. A process that is not stopped will result in an informational return code.

Note that `resume` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the process has resumed or failed to resume.

Return value

The return value for `resume` indicates whether the request to resume process execution was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to resume execution was successfully submitted
`ASC_operation_failed`	resume operation failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully resumed
`ASC_operation_failed`	attempt to resume this process failed

See Also

`attach, battach, bconnect, bdetach, bdisconnect, bresume, bsuspend, connect, detach, disconnect, suspend`

13.54 send_stdin

Synopsis

```
#include <Process.h>

AisStatus send_stdin(char *buffer, int size)
```

Parameters

`buffer`	character array that contains text to be fed to the process stdin
`size`	number of bytes in the buffer to be given to the process

Description

This function provides text to be used as input to the process for the `stdin` device, that is, file descriptor 0. This function is only appropriate for processes that are created using the `create` or `bcreate` member functions.

In order for send_stdin to be used, the Process must have been created using the create function. If a file

Note that `send_stdin` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the process has received the input.

Return value

The return value for `send_stdin` indicates whether the request to provide process input was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to provide input was successfully submitted
`ASC_operation_failed`	request to provide input failed

Callback Data

The acknowlegement callback function is invoked once when the buffer has been sent to the process. When the callback is invoked the callback function is passed a pointer to the Process as the callback object. The callback message is the request status, of type `AisStatus`, which may contain one of the status values values that follow.

`ASC_success`	the buffer was successfully sent to poe
`ASC_operation_failed`	attempt to send the buffer to poe failed

See Also

`bcreate, create`

13.55 set_phase_exit

Synopsis

```
#include <Process.h>
AisStatus set_phase_exit(
     const Phase &ps,
     ProbeExp begin_func,
     GCBFuncType begin_cb_fp,
     GCBTagType begin_cb_tag,
     ProbeExp iter_func,
     GCBFuncType iter_cb_fp,
     GCBTagType iter_cb_tag,
     ProbeExp end_func,
     GCBFuncType end_cb_fp,
     GCBTagType end_cb_tag,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`ps`	phase description to be removed from the application
`begin_func`	initialization function that is executed once within the application when the phase is removed
`begin_cb_fp`	callback function to handle messages from the initialization function
`begin_cb_tag`	tag to be used with the initialization callback function
`iter_func`	iteration function that is executed within the application on each piece of data associated with the phase when the phase is removed
`iter_cb_fp`	callback function to handle messages from the iteration function
`iter_cb_tag`	tag to be used with the iteration callback function
`end_func`	termination function that is executed once within the application when the phase is removed
`end_cb_fp`	callback function to handle messages from the termination function
`end_cb_tag`	tag to be used with the termination callback function
`ack_cb_fp`	callback function to process phase removal acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function specifies a set of exit functions to be executed when any of the following three events occur.

- when the indicated phase is removed using either the remove_phase or bremove_phase function call
- when diconnecting from the target process (without calling remove_phase or bremove_phase first)
- when the target process has finished execution while the indicated phase is still active

Note that `set_phase_exit` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the exit functions have been placed in the indicated phase or the operation failed to complete.

Each of the phase functions must be loaded into the application before this operation may take place. The function prototypes for the functions are:

- `void begin_func(void *msg_handle)`
- `void iter_func(void *msg_handle, void *data)`
- `void end_func(void *msg_handle)`

Return value

The return value for `set_phase_exit` indicates whether the request to set exit functions for the indicated phase on the process was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	remove request was successfully submitted
`ASC_operation_failed`	remove operation failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

begin_cb_fp, **iter_cb_fp**, **end_cb_fp**. These callback functions are invoked each time the corresponding function in the process instrumentation -- `begin_func`, `iter_func`, or `end_func` -- sends a message to the client. The message format is determined by the function that sends the message.

ack_cb_fp. The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	phase was successfully removed from this process
`ASC_operation_failed`	attempt to remove phase from this process failed

See Also

`bset_phase_exit`, `add_phase`, `badd_phase`, `remove_phase`, `bremove_phase`, `set_phase_period`, `bset_phase_period`, `get_phase_period`

13.56 set_phase_period

Synopsis

```
#include <Process.h>
AisStatus set_phase_period(
     const Phase &ps,
     float period,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`ps`	phase to be modified
`period`	new time interval between successive phase activations, in seconds
`ack_cb_fp`	callback function to process phase acknowledgments
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function changes the time interval between successive activations of a phase. The interval change occurs on a process by process basis for all processes within the application. Processes which do not have the phase installed result in an informational return code. Processes that are not connected result in a warning return code.

The new period is represented by a floating-point value. If the value is positive it represents the time interval in seconds. If the value is zero or positive and smaller than the minimum activation time interval, it represents the minimum activation time interval. In both cases the phase is activated immediately upon setting the new interval. If the value is less than zero the phase is disabled immediately, but left in place for possible future reactivation.

Note that `set_phase_period` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the phase period has been set or failed to be set within the process.

Return value

The return value for `set_phase_period` indicates whether the request to set the phase period was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to set the phase period was successfully submitted
`ASC_operation_failed`	set phase period failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	phase period was successfully set
`ASC_operation_failed`	attempt to set the phase period on this process failed

See Also

`bset_phase_period`, `add_phase`, `badd_phase`, `remove_phase`, `bremove_phase`, `set_phase_exit`, `bset_phase_exit`, `get_phase_period`

13.57 signal - LY

Synopsis

```
#include <Process.h>
AisStatus signal(
     int unix_signal,
     GCBFuncType fp,
     GCBTagType tag)
```

Parameters

`unix_signal`	Unix™ signal to be sent to this process
`ack_cb_fp`	callback function to process the signal acknowledgment
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function sends the specified signal to the process. The process must be both connected and attached to receive the signal.

A signal is sent to a process if it is connected and attached.

Note that `signal` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the process has been signaled or failed to be signalled.

Return value

The return value for `signal` indicates whether the request to signal the process was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to signal the processes was submitted
`ASC_operation_failed`	signalling failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully signaled
`ASC_operation_failed`	attempt to signal this process failed

See Also

13.58 start

Synopsis

```
#include <Process.h>
AisStatus start(GCBFuncType ack_cb_fp, GCBTagType ack_cb_tag)
```

Parameters

`ack_cb_fp`	callback function to process a start acknowledgement
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function is currently being designed. This function starts the execution of a process that has been created but has not yet begun execution.

Note that `start` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the application has been started or failed to be started.

Return value

The return value for `start` indicates whether the request to start the process was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to start the application was submitted
`ASC_operation_failed`	start failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully started
`ASC_operation_failed`	attempt to start this process failed

See Also

`bcreate, bstart, create`

13.59 suspend

Synopsis

```
#include <Process.h>
AisStatus suspend(GCBFuncType fp, GCBTagType tag)
```

Parameters

`fp`	callback function to process the suspend acknowledgement
`tag`	tag to be used as an argument to the callback when it is invoked

Description

This function suspends a process that is executing. A tool must be both connected and attached to a process in order to suspend process execution.

Note that `suspend` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the application has been suspended or failed to be suspended.

Return value

The return value for `suspend` indicates whether the request to suspend execution of the process was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to suspend the process was submitted
`ASC_operation_failed`	suspend failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully suspended
`ASC_operation_failed`	attempt to suspend this process failed

See Also

`bresume, bsuspend, resume`

13.60 unload_module

Synopsis

```
#include <Process.h>
AisStatus unload_module(
     ProbeModule *module,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

module	probe module to be unloaded.
ack_cb_fp	callback function to process module removal acknowledgments
ack_cb_tag	tag to be used as an argument to the acknowledgement callback when it is invoked

Description

This function unloads the module from all the processes within the Application class. Onc unloaded, All the probe handles that refer to this probe module are automatically removed.

Note that `unload_module` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the module has been removed or failed to be removed from the process.

Return value

The return value for `unload_module` indicates whether the request to remove the indicated module on the process was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	remove request was successfully submitted
`ASC_operation_failed`	remove operation failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `Ais-Status`, which contains one of the following status values:

`ASC_success`	module was successfully removed from this process
`ASC_operation_failed`	attempt to remove module from this process failed

See Also

`bload_module, bunload_module, load_module`

13.61 writemem - LY

Synopsis

```
#include <Process.h>
AisStatus writemem(
     char *location,
     char *buffer,
     int size,
     GCBFuncType ack_cb_fp,
     GCBTagType ack_cb_tag)
```

Parameters

`location`	address in the application process where writing is to begin
`buffer`	address in the client process from which data is to be taken
`size`	size, in bytes, of both the buffer and the memory block to be written
`ack_cb_fp`	callback function to process a start acknowledgement
`ack_cb_tag`	tag to be used as an argument to the callback when it is invoked

Description

This function sends a request to the daemon managing this process to write the indicated block of memory within the process. Data to write the block of memory is taken from the indicated client buffer.

Note that `writemem` returns control to the caller immediately upon submitting the request to the daemon. It does not wait until the application has been suspended or failed to be suspended.

Return value

The return value for `writemem` indicates whether the request to write data into the memory of the process was successfully submitted. It gives no indication of whether the request was successfully executed.

`ASC_success`	request to write data was submitted
`ASC_operation_failed`	write failed to be requested

Callback Data

When no callback function is provided, that is, when a value of 0 is used as the value for the callback function, the operation is still executed but no callback is called.

The callback function is invoked once, when the acknowledgement of the completion of this operation is received. When the callback is invoked the callback function is passed a pointer to the process as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	data was successfully written to process memory
`ASC_operation_failed`	attempt to write data to this process failed

See Also

`breadmem, readmem, writemem`

14.0 class SourceObj

14.1 Supporting Data Types

14.1.1 Access

Synopsis

```
#include <SourceObj.h>
enum Access {
     SOA_unknown_access,
     SOA_shared,
     SOA_exclusive,
     SOA_LAST_ACCESS
}
```

Description

This enumeration type describes whether the source object to which it applies is part of a shared library or part of a non-shared library.

14.1.2 Binding

Synopsis

```
#include <SourceObj.h>
enum Binding {
     SOB_unknown_binding,
     SOB_static,
     SOB_dynamic,
     SOB_LAST_BINDING
}
```

Description

This enumeration type describes whether the source object to which it applies was bound statically or dynamically by the linker when references to external functions and data were resolved.

14.1.3 LpModel

Synopsis

```
#include <SourceObj.h>
enum LpModel {
     SOL_unknown_model,
     SOL_lp32,
     SOL_lp64,
     SOL_LAST_MODEL
}
```

Description

This enumeration type describes whether the source object to which it applies was compiled and linked with the 32-bit address memory model or the 64-bit address memory model enabled. All objects within a program are compiled and linked with the same model.

14.1.4 SourceType

Synopsis

```
#include <SourceObj.h>
enum SourceType {
     SOT_unknown_type,
     SOT_program,
     SOT_module,
     SOT_function,
     SOT_data,
     SOT_loop,
     SOT_block,
     SOT_statement,
     SOT_LAST_TYPE
}
```

Description

This enumeration type describes whether the source object to which it applies represents a whole program, module, function, data object, *etc*.

14.2 Constructors

Synopsis

```
#include <SourceObj.h>

SourceObj(void)

SourceObj(const SourceObj &copy)
```

Parameters

`copy`	source object that will be duplicated in a copy constructor

Description

The default constructor creates an empty source object whose access, binding, LP model and source type are each set to “unknown”. The default constructor is invoked when uninitialized source objects are created, such as in arrays of source objects. Objects within the array can be overwritten using an assignment operator (`operator =`).

The copy constructor is used to transfer the contents of an initialized object (the `copy` parameter) to an uninitialized object.

Exceptions

`ASC_insufficient_memory`	not enough memory to create a new node

See Also

14.3 address_end

Synopsis

```
#include <SourceObj.h>

void *address_end(void) const
```

Description

This function returns the virtual address of the last element associated with this source object. If the source object represents a scalar data object, then `start_address` and `end_address` return the same value. If the source object represents an array, then it returns the virtual address of the last element in the array. If the source object represents a function, then it returns the approximate address of the last instruction in the function.

Return value

Virtual address of the last element associated with this source object

See Also

14.4 address_start

Synopsis

```
#include <SourceObj.h>

void *address_start(void) const
```

Description

This function returns the virtual address of the first element associated with this source object. If the source object represents a scalar data object, then `start_address` and `end_address` return the same value. If the source object represents an array, then it returns the virtual address of the first element in the array. If the source object represents a function, then it returns the approximate address of the first instruction in the function.

Return value

Virtual address of the first element associated with this source object

See Also

14.5 bexpand

Synopsis

```
#include <SourceObj.h>
AisStatus bexpand(const Process &proc)
```

Parameters

`proc`	process to which the “expand” request applies

Description

This function applies only to source objects with `SourceType` of `SOT_module`. The function requests that the details of an unexpanded module be supplied. Modules are not expanded when the client initially connects with a process. Modules that are not expanded cannot be examined for additional structure, such as data, functions, and instrumentation points. Recommended use is to establish a connection to a process, then expand those modules where one wishes to place instrumentation.

If the `SourceType` is not `SOT_module`, the function immediately returns with a status of `ASC_operation_failed`.

Note that the function submits the request to expand the source object and waits until the request has completed.

Return value

The return value indicates whether the request for expansion was successfully executed.

`ASC_success`	expansion was successfully completed
`ASC_operation_failed`	expansion failed

See Also

14.6 child

Synopsis

```
#include <SourceObj.h>

SourceObj child(int index) const
```

Parameters

`index`	index into the source object child table, which must be greater than or equal to zero, and less than `child_count()`

Description

This function returns the child indicated by the parameter `index`. `Index` must be greater than or equal to zero, and less than `child_count()`. When `child()` is given an index value that is outside of this range, it returns an empty source object, as created by the default constructor. Children can be variables, functions, modules, *etc*.

Return value

Child source object indicated by the parameter `index`.

See Also

14.7 child_count

Synopsis

```
#include <SourceObj.h>

int child_count(void) const
```

Description

This function returns the number of child source objects associated with this source object. Empty source objects, created by the default constructor, return zero. Children can be variables, functions, modules, *etc*.

Return value

Number of child source objects associated with this source object.

See Also

14.8 exclusive_point

Synopsis

```
#include <SourceObj.h>
InstPoint exclusive_point(int index) const
```

Parameters

`index`	index into the instrumentation point table, which must be greater than or equal to zero, and less than `exclusive_point_count()`.

Description

This function returns the instrumentation point indicated by the parameter `index`. Instrumentation points contained only within this source object are arranged in a table whose smallest index is 0 and whose largest index is `exclusive_point_count()-1`.

Return value

Instrumentation point indicated by the parameter `index`.

See Also

`exclusive_point_count`

14.9 exclusive_point_count

Synopsis

```
#include <SourceObj.h>

int exclusive_point_count(void) const
```

Description

This function returns the number of instrumentation points associated with only this source object.

Return value

Number of instrumentation points associated with this source object.

See Also

`exclusive_point`

14.10 expand

Synopsis

```
#include <SourceObj.h>
AisStatus expand(Process proc, GCBFuncType fp, GCBTagType tag)
```

Parameters

`proc`	process to which the “expand” request applies

Description

This function applies only to source objects with `SourceType` of `SOT_module`. The function requests that the details of an unexpanded module be supplied. Modules are not expanded when the client initially connects with a process. Modules that are not expanded cannot be examined for additional structure, such as data, functions, and instrumentation points. Recommended use is to establish a connection to a process, then expand those modules where one wishes to place instrumentation.

If the `SourceType` is not `SOT_module`, the function immediately returns with a status of `ASC_operation_failed`.

Note that the function submits the request to expand the source object and returns immediately. It does *not* wait until the request has completed.

Return value

The return value for `expand` indicates whether the request was successfully submitted, but indicates nothing about whether the request itself was successfully executed.

Callback Data

The callback function is invoked once for each expansion request. When the callback is invoked the callback function is passed a pointer to the source object as the callback object. The callback message is the request status, of type `AisStatus`, which contains one of the following status values:

`ASC_success`	process was successfully attached
`ASC_operation_failed`	attempt to attach to this process failed

See Also

14.11 get_access

Synopsis

```
#include <SourceObj.h>
Access get_access(void) const
```

Description

This function returns the access type of the source object, that is, whether it is part of a shared library or not. Functions within a shared library are marked as `SOA_shared`. All others are designated `SOA_exclusive`. All variables are private to a program, even those in shared libraries, and are therefore marked `SOA_exclusive`.

Return value

`SOA_shared`	object is a function from a shared library
`SOA_exclusive`	object is not from a shared library, or it is data
`SOA_unknown`	uninitialized object

See Also

14.12 get_binding

Synopsis

```
#include <SourceObj.h>

Binding get_binding(void) const
```

Description

This function returns the binding type of the object. The binding type refers to whether the function or module is part of a dynamically loaded library. When it is part of a dynamic library `get_binding` returns `SOB_dynamic`. Otherwise it returns `SOB_static`.

Return value

`SOB_dynamic`	object is from a dynamically loaded library
`SOB_static`	object is not from a dynamically loaded library
`SOB_unknown`	uninitialized object

See Also

14.13 get_data_type

Synopsis

```
#include <SourceObj.h>
ProbeType get_data_type(void) const
```

Description

This function returns the data type of the object when the object represents a function or a variable. When the object represents something that is neither a function nor a variable, it returns a data type tagged as "unknown".

Return value

Data type of the object, or "unknown".

See Also

14.14 get_demangled_name

Synopsis

```
#include <SourceObj.h>
char *get_demangled_name(char *buffer, unsigned int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the demangled name
`len`	maximum number of bytes the function will place in *buffer*. The *len* parameter should include enough space for a terminating *null* byte.

Description

This function copies into *buffer* a *null*-terminated string representing the demangled name of an object when that object is a function. The name may be truncated if the *len* parameter is smaller than the length of the demangled name. A function's demangled name is the name of a function as it appears in the original source code of a program as seen by a compiler.

Return value

Pointer to *buffer*, which will contain at most *len* bytes of the demangled function name when the object is a function; 0 otherwise.

See Also

`get_mangled_name, get_demangled_name_length`

14.15 get_demangled_name_length

Synopsis

```
#include <SourceObj.h>
unsigned int get_demangled_name_length(void) const
```

Description

This function returns the length, including the terminating *null* byte, of the demangled name of a function.

Return value

When the object is a function, the length of the object's demangled name; 0 otherwise.

See Also

`get_demangled_name`

14.16 get_mangled_name

Synopsis

```
#include <SourceObj.h>

char *get_demangled_name(char *buffer, unsigned int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the demangled name
`len`	maximum number of bytes the function will place in *buffer*. The *len* parameter should include enough space for a terminating *null* byte.

Description

This function copies into *buffer* a *null*-terminated string representing the mangled name of an object when that object is a function. The name may be truncated if the *len* parameter is smaller than the length of the mangled name. A function's mangled name is the name of a function as it appears to the linker and loader. Name mangling is supported by compilers and linkers to resolve overloaded function names in object-oriented programming languages. In order to distinguish between two functions that have the same programmer-visible name, compilers encode parameter type information into the actual function name as it is seen by the linker and loader.

Mangled names include parameter data type information for some languages, notably C++ and Fortran 90, but not necessarily for all languages.

Return value

Pointer to *buffer*, which will contain at most *len* bytes of the mangled function name when the object is a function; 0 otherwise.

See Also

`get_demangled_name, get_mangled_name_length`

14.17 get_mangled_name_length

Synopsis

```
#include <SourceObj.h>
unsigned int get_demangled_name(void) const
```

Description

This function returns the length, including the terminating *null* byte, of the mangled name of a function.

Return value

When the object is a function, the length of the object's mangled name; 0 otherwise.

See Also

```
get_mangled_name
```

14.18 get_program_type

Synopsis

```
#include <SourceObj.h>
LpModel get_program_type(void) const
```

Description

This function returns an indicator of whether the program is using the 32-bit address memory model, or the 64-bit address memory model. All functions within a program must use the same memory model. AIX does not support mixed address models.

Return value

`SOL_lp32`	program uses the 32-bit address memory model
`SOL_lp64`	program uses the 64-bit address memory model
`SOL_unknown`	uninitialized object

See Also

14.19 get_variable_name

Synopsis

```
#include <SourceObj.h>

char *get_variable_name(char *buffer, int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the variable name
`len`	maximum number of bytes the function will place in *buffer*. The *len* parameter should include enough space for a terminating *null* byte.

Description

This function copies into *buffer* a *null*-terminated string representing the name of the object when the object is a data variable. To check if an object is a data variable, look for a return type of SOT_data from src_type(). The name may be truncated if the *len* parameter is smaller than the length of the variable name.

Return value

If the object is a data variable, a pointer to *buffer*, which will contain at most *len* bytes of the name.

0 if the object is not a data variable..

See Also

`get_variable_name_length, src_type`

14.20 get_variable_name_length

Synopsis

```
#include <SourceObj.h>
unsigned int get_variable_name_length(void) const
```

Description

This function returns the length, including the terminating *null* byte, of the name of the object when the object is a data variable.

Return value

If the object is a data variable, the length of the name.

0 if the object is not a data variable..

See Also

`get_variable_name`

14.21 inclusive_point

Synopsis

```
#include <SourceObj.h>
InstPoint inclusive_point(int index) const
```

Parameters

`index` index into the instrumentation point table, which must be greater than or equal to zero, and less than `inclusive_point_count()`.

Description

This function returns the instrumentation point indicated by the parameter `index`. All instrumentation points contained within this source object and its children are arranged in a table whose smallest index is 0 and whose largest index is `inclusive_point_count()-1`.

Return value

Instrumentation point indicated by the parameter `index`.

See Also

`inclusive_point_count`

14.22 inclusive_point_count

Synopsis

```
#include <SourceObj.h>
int inclusive_point_count(void) const
```

Description

This function returns the number of instrumentation points associated with this source object and all of its children.

Return value

Number of instrumentation points associated with this source object and all of its children.

See Also

`inclusive_point`

14.23 library_name

Synopsis

```
#include <SourceObj.h>
char *library_name(char *buffer, unsigned int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the library name
`len`	maximum number of bytes the function will place in *buffer*. The *len* parameter should include enough space for a terminating *null* byte.

Description

This function copies into *buffer* a *null*-terminated string representing the name of the library that contains the object. The name may be truncated if the *len* parameter is smaller than the length of the library name.

Return value

A pointer to *buffer*, which will contain at most *len* bytes of the library name.

0 if the object is not contained within a library or when the information has been removed from the executable.

See Also

`library_name_length`

14.24 library_name_length

Synopsis

```
#include <SourceObj.h>
unsigned int library_name_length(void) const
```

Description

This function returns the length, including the terminating *null* byte, of the string representing the name of the library that contains the object.

Return value

The length of the library name.

0 if the object is not contained within a library or when the information has been removed from the executable.

See Also

`library_name`

14.25 line_end

Synopsis

```
#include <SourceObj.h>
int line_end(void) const
```

Description

This function returns the approximate line number of the last line in the object. When the line number is unknown or undefined, the function returns -1.

Return value

Approximate line number of the last line in the object, or -1.

See Also

14.26 line_start

Synopsis

```
#include <SourceObj.h>

int line_start(void) const
```

Description

This function returns the approximate line number of the first line in the object. When the line number is unknown or undefined, the function returns -1.

Return value

Approximate line number of the first line in the object, or -1.

See Also

14.27 module_name

Synopsis

```
#include <SourceObj.h>

char *module_name(char *buffer, unsigned int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the module name
`len`	maximum number of bytes the function will place in *buffer*. The *len* parameter should include enough space for a terminating *null* byte.

Description

This function copies into *buffer* a *null*-terminated string representation of the file name and path of the module that contains the object. The name may be truncated if the *len* parameter is smaller than the length of the module name..

Return value

A pointer to buffer, which will contain the file name and path of the module that contains this object.

0 if the object is the program object, which.is not contained within any module

See Also

`module_name_length`

14.28 module_name_length

Synopsis

```
#include <SourceObj.h>
unsigned int module_name(void) const
```

Description

This function returns the length, including the terminating *null* byte, of the file name and path of the module that contains the object.

Return value

The length of the file name and path of the module that contains this object.

0 if the object is the program object, which.is not contained within any module

See Also

`module_name`

14.29 obj_parent

Synopsis

```
#include <SourceObj.h>

SourceObj obj_parent(void) const
```

Description

This function returns the parent object of this object. For example, the parent object of a function object is a module object. The parent object of a program object is itself.

Return value

Parent object of the object.

See Also

14.30 operator =

Synopsis

```
#include <SourceObj.h>
SourceObj &operator = (const SourceObj &copy)
```

Parameters

`copy` source object to be duplicated

Description

This function transfers the contents of the `copy` parameter to the object.

Return value

Reference to the object.

See Also

14.31 operator ==

Synopsis

```
#include <SourceObj.h>

int operator == (const SourceObj &compare)
```

Parameters

`compare`	source object to be compared

Description

This function compares two source objects for equivalence. If the two objects represent the same portion of the program or application, this function returns 1. Otherwise it returns 0.

Return value

This function returns 1 if the two objects are equivalent, 0 otherwise.

See Also

14.32 operator !=

Synopsis

```
#include <SourceObj.h>
int operator != (const SourceObj &compare)
```

Parameters

`compare` source object to be compared

Description

This function compares two source objects for equivalence. If the two objects represent the same portion of the program or application, this function returns 0. Otherwise it returns 1.

Return value

This function returns 0 if the two objects are equivalent, 1 otherwise.

See Also

14.33 program_name

Synopsis

```
#include <SourceObj.h>
char *program_name(char *buffer, unsigned int len) const
```

Parameters

`buffer`	caller-allocated buffer to hold the program name
`len`	maximum number of bytes the function will place in *buffer*. The *len* parameter should include enough space for a terminating *null* byte.

Description

This function copies into *buffer* a *null*-terminated string representing the file name and path of the executable program (`a.out`). The name may be truncated if the *len* parameter is smaller than the length of the program name.

Return value

A pointer to buffer, which will contain the file name and path of the executable.

0 if this information is not available.

See Also

`program_name_length`

14.34 program_name_length

Synopsis

```
#include <SourceObj.h>
unsigned int program_name_length(void) const
```

Description

This function returns the length of the file name and path of the executable program (`a.out`).

Return value

The length of the file name and path of the executable.

0 if this information is not available.

See Also

`program_name`

14.35 ref_to_probe_exp

Synopsis

```
#include <SourceObj.h>

ProbeExp ref_to_probe_exp(void) const
```

Description

This function creates a reference to a program function or variable that may be used in a probe expression. References to program functions may be used in creating calls to those functions, while references to program variables may be used to read, modify, or write those variables. When the object does not represent a program function or variable, an “undefined” probe expression is returned. To see if a SourceObj is a program function or variable, use src_type() and check for return types of SOT_function or SOT_data.

Return value

Reference to the program function or data, or an “undefined” probe expression.

See Also

`src_type`

14.36 src_type

Synopsis

```
#include <SourceObj.h>
SourceType src_type(void) const
```

Description

This function returns the type of source object represented by the object. The source object type corresponds to various objects within a program, such as modules, functions, variables, *etc*. If the source object does not correspond to a program or part of a program, the source object type is “unknown”.

Return value

Type of this source object.

See Also

```
get_variable_name
```

15.0 Miscellaneous Functions

15.1 Ais_initialize

Synopsis

```
#include <AisInit.h>

void Ais_initialize(void)
```

Description

This function is used to control the initialization and re-initialization of certain sub-systems, such as the registration of internal callbacks, within the instrumentation system. It must be called once before entering the main event loop.

See Also

15.2 Ais_end_main_loop

Synopsis

```
#include <AisMainLoop.h>
void Ais_end_main_loop(void)
```

Description

This function is used to indicate to the main event loop that processing is to be terminated, and no more events are to be consumed. It does not cause any connections to be lost, nor to be closed. It only terminates the event processing loop that gathers event messages from all connected daemons.

See Also

`Ais_main_loop`

15.3 Ais_main_loop

Synopsis

```
#include <AisMainLoop.h>
void Ais_main_loop(void)
```

Description

This function is the main event loop for the instrumentation system. This loop processes events in the form of special messages from daemons and instrumented processes. It must be called after the initialization function. It must be called in order for the instrumentation system to process events and messages from the application processes. This function does not return control to the caller until `Ais_end_main_loop() is called.`

See Also

`Ais_end_main_loop`

15.4 Ais_override_default_callback

Synopsis

```
#include <AisHandler.h>
AisStatus Ais_override_default_callback(
    unsigned msg_type,
    GCBFuncType new_cb_fp,
    GCBTagType new_cb_tag,
    GCBFuncType &old_cb_fp,
    GCBTagType &old_cb_tag)
```

Parameters

`msg_type`	message type for which the callbacks are to be overridden
`new_cb_fp`	callback function to be invoked when messages of the specified type are received
`new_cb_tag`	tag to be used with the new callback function
`old_cb_fp`	callback function previously registered for this message type
`old_cb_tag`	tag previously registered for this message type

Description

This function allows the caller to replace the callback chain associated with the specified message type with a new callback function. When the client receives a message from a daemon or other message source that uses the messaging/callback system there is a message type identifier associated with the message. This message type identifier is used as a key for looking up a callback chain to be executed as a result of receiving that message. The message itself, information contained within the message envelope, the tag, and other information are passed to each function in the callback chain.

Return value

The return value indicates whether the attempt to override the callback chain was successful.

`ASC_success`	message type is registered and the callback chain was updated
`ASC_???`	message type is not registered

See Also

`AIS_EXIT_MSG, AIS_ERROR_MSG, AIS_OUTPUT_MSG, AIS_DEFAULT_CB`

16.0 Predefined Global Variables

16.1 AIS_DEFAULT_CB

Synopsis

```
#include <AisMsgType.h>
extern const int AIS_DEFAULT_CB
```

Description

This constant represents a callback identifier key. This callback chain is used when a message is received that has no callback chain for the message type. A tool may alter the callback function associated with this key with the `Ais_override_default_callback` function.

See Also

16.2 AIS_ERROR_MSG

Synopsis

```
#include <AisMsgType.h>
extern const int AIS_ERROR_MSG
```

Description

This constant represents a callback identifier key. It may be used by daemon processes to send an error message to the end user. A tool may alter the callback function associated with this key with the `Ais_override_default_callback` function.

See Also

16.3 AIS_EXIT_MSG

Synopsis

```
#include <AisMsgType.h>
extern const int AIS_EXIT_MSG
```

Description

This constant represents a callback identifier key. It may be used by daemon processes to send an exit message to the end user. A tool may alter the callback function associated with this key with the `Ais_override_default_callback` function.

See Also

16.4 Ais_msg_handle

Synopsis

```
#include <AisGlobal.h>

extern const ProbeExp Ais_msg_handle
```

Description

This constant represents a probe-specific value that is used to send messages from the probe to the client. Each probe is able to send messages to the client any time the probe is invoked. The client is able to distinguish between messages from one probe and messages from another. Furthermore, more than one client can be connected to an application process, and the probe must maintain some record of the client to whom it belongs. All the necessary information to accomplish these things is stored in the probe message handle. The probe message handle is used as the first argument to the `Ais_send` function, that sends a message to the client, to be processed by a client data callback function.

See Also

16.5 Ais_send

Synopsis

```
#include <AisGlobal.h>

extern const ProbeExp Ais_send
```

Description

This constant represents a function that allows probes to send messages to the client. The function may be executed directly by the probe as any other function. The type signature for the send function is:

```
void Ais_send( void *msg_handle, char *buffer, int size )
```

where `msg_handle` is the constant `Ais_msg_handle`, `buffer` is the message to be sent, and `size` is the number of bytes in the message.

See Also

```
ProbeExp::call
```

16.6 AIS_OUTPUT_MSG

Synopsis

```
#include <AisMsgType.h>

extern const int AIS_OUTPUT_MSG
```

Description

This constant represents a callback identifier key. It may be used by daemon processes to send a message to the end user. A tool may alter the callback function associated with this key with the `Ais_override_default_callback` function.

See Also

16.7 AIS_PROC_TERMINATE_MSG

Synopsis

```
#include <AisMsgType.h>

extern const int AIS_PROC_TERMINATE_MSG
```

Description

This constant represents a callback identifier key. It may be used by daemon processes to send a message to the end user indicating an application process has terminated. A tool may alter the callback associated with this key with the `Ais_override_default_callback` function.

See Also

Index