Application Development with the OPUS Application Programming Interface (OAPI).

OPUS v2.2

Data Processing Team, Engineering & Software Services Division, Space Telescope Science Institute, Baltimore, Maryland, USA

Preface....1
OPUS....1
OAPI....2
Blackboard, Entry, and Field Class Hierarchies....4
  Blackboard Class Hierarchy....4
  Entry Class Hierarchy....5
  Field Class Hierarchy....5
Opus_env Class....6
Event Handling....7
Exception Handling....9
Opus_lock Class Hierarchy....11
Utility Classes: Oresource & Pipeline....11
Utility Classes: Msg....12
Utility Classes: Ofile, Opus_pid, Num_in_str....12
Memory Management....13
The Clone Idiom....13
Resource Locking....14
Compiling & Linking Against the OAPI....15
A File Poller....18
A Simple "Collector"....20
Non-Pipeline Application....26

Chapter 1

Introduction

Preface

This document is intended to serve as a guide for OPUS users that want to develop their own pipeline applications using the OPUS Application Programming Interface (OAPI). A basic knowledge of OPUS pipelines and terminology are assumed. If you are new to OPUS, it is recommended that you first read the OPUS FAQ and experiment with the sample pipeline distributed with OPUS. Additional information is available on the OPUS Home Page (http://www.stsci.edu/opus).

As you proceed through this guide, keep in mind that it is just that—a guide and not a complete reference manual. A reasonable attempt is made to provide you with the high-level design philosophy behind the OAPI to aid you in designing your own OPUS applications. You will find brief descriptions of some its major components, and a few examples of OPUS applications that are not too different from applications we have developed. Most certainly, you will need to consult the OAPI HTML documentation for specific information on using the OAPI classes when you set off to write your own code.

OPUS

The OPUS platform is a distributed pipeline system that allows multiple instances of multiple processes to run on multiple nodes over multiple paths. While OPUS was developed to support the telemetry processing for the HST instruments, it is a generic pipeline system, and is not tied to any particular processing environment, or to any particular mission. From this point of view the OPUS platform does not provide the mission specific applications themselves. Instead OPUS provides a fully distributed pipeline processing environment structured to help organize the applications, monitor the processing and control what is going on in the pipeline.

The basic architecture of the OPUS system is based on a blackboard model where processes do not communicate directly with one another, but simply read and write to a common “blackboard”. In the current implementation of OPUS, the blackboards are accessed through the (network) file system as a directory on a commonly accessible disk. In a cluster of workstations and larger machines, if the protections are set appropriately, any process can "see" any file in the blackboard directory: the "posting" of blackboard messages consists of either creating or renaming an empty file in that directory.

An OPUS pipeline is defined by the set of applications that processes data or that performs tasks in a co-operative manner and the rules that determine when they should act and how their results should be interpreted. OPUS pipeline applications fall into two general classes: *internal* pollers and *external* pollers. Internal pollers are developed with explicit knowledge of the OPUS environment—they make direct calls into the OAPI library

for initialization and for event handling, and must be linked against the OAPI library. External pollers use a proxy application (**xpoll**[1]) to communicate with OPUS and typically are wrapped by a shell script. **xpoll** interacts directly with OPUS and executes the external poller whenever work is to be performed by that process. **xpoll** communicates OPUS event data and receives process completion status through the external poller's environment. As long as a suitable shell script can be developed that meets the input requirements of the application, any application can be used in an OPUS pipeline. Both internal and external pollers share many of the same basic capabilities although internal pollers, by virtue of having direct access to the OAPI, are more flexible. The OPUS Sample Pipeline demonstrates both classes of applications (**g2f** is an internal poller; all of the other applications are external pollers).

OAPI

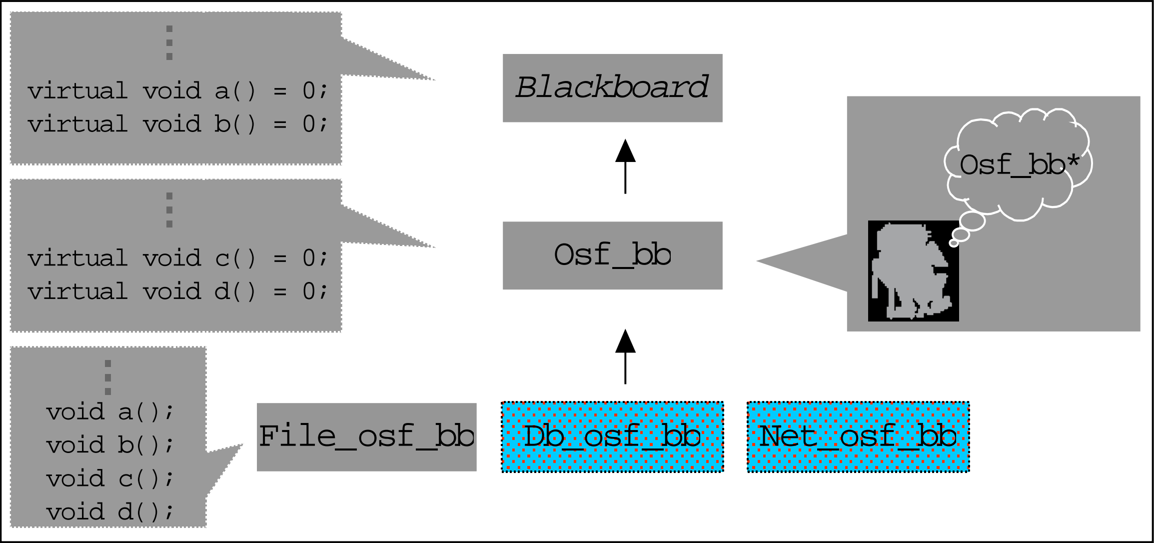
The OPUS Application Programming Interface (OAPI) is an object-oriented, C++ interface to the OPUS environment distributed for Solaris[2], Linux[3], and Tru64[4] platforms. With the OAPI, internal-polling OPUS pipeline applications can be developed that take full advantage of the capabilities and flexibility offered by OPUS. The OAPI contains classes for interacting with the OPUS blackboards and their contents, for reading an assortment of resource files, for message reporting, for event handling, and for exception handling. Its functionality can be extended to include additional or customized features not yet provided by the library through traditional object-oriented techniques like inheritance and composition.

The OAPI was designed to satisfy the needs of two groups of software developers. On the one hand, it serves the programmer who wants to develop OPUS-savvy processes without regard for the implementation details of the OPUS system. Ease of use is a primary consideration for such a developer. On the other hand, it must be easily maintainable, backwards compatible with previous versions of OPUS, and offer the flexibility to meet future requirements of OPUS pipelines with little impact on existing code. These goals demand a general, abstract approach to the architecture with strict isolation of interface from implementation--a methodology that is often at odds with ease of use. The library follows a middle-of-the-road tack by promoting flexibility and ease of use through run-time polymorphism.

The OAPI exposes a set of interfaces defined by a set of core abstract base classes. Where applicable, the base classes provide an implementation, but far more often, specialized classes are derived from these base classes that provide the actual functionality of OPUS. Access to these derived types is achieved through a pointer to the base class and is transparent to the client. Using the run-time polymorphic behavior of C++ class hierarchies in this way is a powerful tool that helps preserve a high degree of separation between implementation and interface. Separating implementation from interface allows the use of generic algorithms to process different implementations of an object through a common interface thereby reducing code duplication and development effort. In addition, it permits evolution of the OAPI with minimal impact on the clients of the

[1] **xpoll** is short for e**X**ternal **POLL**er and is itself an internal polling process.
[2] Developed under Solaris 2.7 using the EGCS 1.1.2 release.
[3] Developed under RedHat Linux 6.1 using the EGCS 1.1.2 release.
[4] Developed under Tru64 UNIX V5.0A using Compaq C++ v6.2-024 (template repository included) and Compaq C V6.1-013.

library since they only reference the interface exposed by the base classes. Figure 1 illustrates how run-time polymorphism is used by the OAPI to isolate client code from the implementation details in the library.



- Figure 1: OAPI Run-Time Polymorphism. Class Blackboard defines an abstract interface to all OPUS blackboards. Class Osf_bb inherits from Blackboard and adds additional items specific to OSF blackboards to the interface; clients of the OAPI manipulate pointers to this type whereas the actual functionality of the OSF blackboard is implemented in either class File_osf_bb, Db_osf_bb, or Net_osf_bb depending on which type of OSF blackboard is in use.

Chapter 2

OAPI Contents: An Overview

Blackboard, Entry, and Field Class Hierarchies

Most classes in the OAPI abstract some aspect of a "blackboard". A blackboard in this context is any class that implements (at least) the interface defined by `Blackboard`. In OPUS, blackboards are imagined to contain lists of unique[5] entries of the same type and the member functions to manage them. The entries on a blackboard inherit from class `Entry` and are composed of a set of fields; the fields are inherited from class `Field`.

Blackboard Class Hierarchy

The Blackboard class hierarchy defines objects that implement message bulletin boards. Blackboards, at their most basic, are containers of entries in this design. Member functions exist for posting an entry on the blackboard, erasing an entry from the blackboard, replacing one entry with another, obtaining a lock on an entry, and searching for entries matching an entry template.

Blackboard Type	Description
`Blackboard`	The abstract base class for all OPUS blackboards. An object of this type cannot be instantiated; it serves solely as an interface definition for all derived classes.
`Command_bb`	A blackboard containing commands to execute (i.e., functions to call).
`Absolute_time_bb`	The blackboard used by OPUS time pollers that specify an absolute start time (i.e., define a `START_TIME` resource keyword).
`Relative_time_bb`	The blackboard used by OPUS time pollers that specify a periodic trigger time (i.e., define a `DELTA_TIME` resource keyword).
`Pstat_bb`	The base class for all OPUS PSTAT blackboard implementations. It defines an interface that all PSTAT blackboards should implement in addition to the Blackboard interface. At present, there is only a file-

[5] Entry uniqueness is determined by `operator==` applied for one or more of its fields.

	name based implementation (see File_pstat_bb).
File_pstat_bb	An implementation of a PSTAT blackboard that stores the PSTAT fields in a file name on the file system.
Osf_bb	The base class for all OPUS OSF blackboard implementations. It defines an interface that all OSF blackboards should implement in addition to the Blackboard interface. At present, there is only a file-name based implementation (see File_osf_bb).
File_osf_bb	An implementation of an OSF blackboard that stores the OSF fields in a file name on the file system.
Files_bb	A blackboard interface to the file system, primarily used by OPUS file pollers.
Pstat_event_bb	A blackboard used to cache the application's PSTAT for performance purposes.

Entry Class Hierarchy

The Entry class hierarchy defines the objects stored on the various blackboards. The base class provides a variety of member functions that deal with the contained fields; classes derived from Entry often augment those basic member functions.

The mapping of blackboard type to entry type is as follows:

Blackboard Type	Entry Type Used on that Blackboard
Command_bb	Command
Absolute_time_bb Relative_time_bb	Time_entry
File_pstat_bb‡	File_pstat‡
File_osf_bb‡	File_osf‡
Files_bb	File_entry

‡ These types normally are referenced indirectly through a pointer to the base type (see Figure 1).

Field Class Hierarchy

The Field class hierarchy defines the fields that make up the various Entry objects.

The mapping of entry type to contained field type is as follows:

Entry Type	Fields Contained by that Entry
Command	Com_label, Com_arg
Time_entry	Delta_time or Absolute_time
Pstat	Pid, Process, Proc_stat, Start_time, Path, Node, Proc_cmd
Osf	Time_stamp, Obs_stat, Dataset, Data_id, Dcf_num, Obs_cmd
File_entry	Directory, Rootname, Extension, Dangle

Opus_env Class

Class Opus_env establishes the OPUS environment for an application. The constructor performs command line argument parsing and creation of the appropriate blackboards. A typical internal polling process that uses the OAPI instantiates one of these objects immediately after entering main. Opus_env also acts as a convenient interface to commonly called OPUS facilities once the process is up and running in the pipeline. For example, the get_res_item member function returns a keyword value from the process resource file after performing path file value substitution as necessary.

An important item to keep in mind is that only one instance of Opus_env can be instantiated per process. To access any of its member functions, a reference or pointer to the object must be in scope or an appropriate singleton defined. The callback functions for OPUS events automatically receive a reference to Opus_env to aid in gaining access to the object.

After constructing the Opus_env object and performing any other one-time initialization tasks, a pipeline application typically enters a polling loop in which the poll member function is repeatedly called followed by processing of the returned event. A simple code fragment that demonstrates initialization and the polling loop follows:

```
Opus_env opus(argc, argv);     // Initialize OPUS
if (!opus.is_initialized()) { // Construction failed- exit.
       cerr << "Bailing out." << endl;
       exit(EXIT_FAILURE);
}

Halt_event::add_callback(halt_handler); // register callback for
                                        // HALT events
while(true) {
       Event* e = opus.poll(); // poll for an event
       try {
          e->process();        // process event (callback is called)
```

```
        }
        catch(...) {
               cerr << "A problem was encountered processing: " <<
               e->trigger_name() << endl;
               exit(EXIT_FAILURE);
        }
        delete e;               // dispose of the event
}
```

In this example, a callback function to handle HALT events is registered (the code for the function `halt_handler` is not shown), and then the polling loop is entered. Should the call to `Opus_env::poll()` return a HALT event, the `halt_handler` function is called by the `process` member function of the `Event` object. The next section describes the types of events the library can generate and to which an application can respond.

Applications like `osf_create` that are not run in the pipeline, but that require access to the OPUS blackboards, also can use `Opus_env` with a special form of the constructor. The last case study in Chapter 4 provides an example of such an application and illustrates this specialized use.

Event Handling

Once polling of the blackboards is initiated via a call to `Opus_env::poll`, the application enters an event-driven mode where callbacks are used to handle events as they occur. The events may be user-defined (through OSF, file, and time triggers defined in the process resource file) or they may result from external signals sent to the process (a HALT or REINIT command issued by the PMG, for example). In addition, the `Opus_env` object handles some events internally without ever calling application code (examples include SUSPEND commands issued by the PMG and MINBLOCKS disk space checks).

Application code must register callback functions for each event type it wishes to handle (the default action for an event with no callback is to ignore it). The following event types are defined (note that some of the events are handled by `Opus_env` and callbacks should not be registered for them when using `Opus_env`):

Event Type	Event Trigger	Means of Handling Event
`File_event`	One or more matches to the file mask(s) specified in a process resource file.	Register callback.
`Time_event`	The process resource file specified absolute time or elapsed time has expired.	Register callback.
`Osf_event`	One or more matches to the OSF mask(s) specified in the process resource files.	Register callback.
`Command_event`	A function associated with a `Command` on the `Command_bb` returned `true` when called during a search of the	Register callback.

	blackboard.	
Halt_event	A HALT command issued on the application's PSTAT.	Register callback.
Reinit_event	A REINITIALIZE command issued on the application's PSTAT.	Register callback.
Low_store_event	A check for MINBLOCKS indicates not enough free space on disk.	Handled internally by Opus_env; creation of the event causes an exception of type Internal_event to be thrown that is caught by Opus_env::poll.
Suspend_event	A SUSPEND command issued on the application's PSTAT.	Handled internally by Opus_env; creation of the event causes an exception of type Internal_event to be thrown that is caught by Opus_env::poll.
Suspended_event	Application's PSTAT indicates that it is suspended.	Handled internally by Opus_env; creation of the event causes an exception of type Internal_event to be thrown that is caught by Opus_env::poll.
Resume_event	A RESUME command issued on the application's PSTAT.	Handled internally by Opus_env; creation of the event causes an exception of type Internal_event to be thrown that is caught by Opus_env::poll.

To register a callback function for an event, use the static member function `add_callback` for that event type. Only one callback can be registered per event type. The callback function must have the signature `void func(const string&, Event*, const Opus_env&)`. The first argument is a reference to a string containing the trigger name, the second argument is a pointer to the actual event object, and the third argument is a reference to the `Opus_env` object.

The callback function is called indirectly by the `process` member function of the event in question (usually called in the application's polling loop). The body of the callback function has access to both the event member functions and those of `Opus_env`, and should perform any processing required in order to handle the event. Once processing is complete, the event should be closed by a call to `Opus_env::close_event` *after obtaining locks on any entries in the event that will be updated*. This member function applies any process resource file defined modifiers to the entries that triggered the event

on the blackboard (for example, an OSF stage might be changed from `p` to `c` to signal successful completion).

Exception Handling

The OAPI takes advantage of C++ exception handling by "passing the buck" when a situation arises during execution of the library code that is best handled by the calling routine. The C++ exception-handling mechanism is of great benefit to the library developer because it is rare that the library code knows exactly what should be done when a run-time error occurs. Moreover, error codes returned by a library are too easily ignored and lead to further library corruption as they go undetected and additional library calls are made with corrupted data or bad state information. On the contrary, exceptions force the caller to take action lest the program abort.

Every exception thrown by the OAPI is derived from class `Opus_exceptions`. Thus, all OAPI exceptions can be caught as a reference to this type of object. Each class in this hierarchy is associated with a loosely defined category of exceptional conditions, and is a class template instantiated for the type of object responsible for the exception being thrown. The table below lists the class templates derived from `Opus_exceptions`.

Class Template	Exceptional Condition
`Bad_val`	A bad or inappropriate object was encountered during processing.
`No_entry`	An item expected to exist in a group of items is missing.
`Type`	The wrong type of object was used in an operation.
`Io_error`	An error occurred during an I/O operation.
`Ambiguous`	An operation resulted in an ambiguous situation that cannot be resolved by the library.
`Severe`	An unexpected error has prevented completion of the operation.
`Already`	An operation was requested that already is complete.
`Not_ready`	A required input for completing the requested operation is not ready or available.
`Exec`	Execution of a sub-process or shell failed.
`File_action`	Application of a field modifier to a `File_entry` object has signaled the need for a `FILE_ACTION` to be

	performed (only used by `Opus_env`).
`Corrupt`	Information required to complete processing appears to be corrupt.
`Locked`	The requested resource is locked by another entity and cannot be reserved.

In general, the object thrown is constructed with the type that caused the exception, or with information useful to the exception handler. For example, if an integer argument passed into a function were out of range, that function might throw an exception of type `Bad_val<int>` where the template argument is the bad integer value. The code producing the exception might look like:

```
if ( i < 0 ) {
       string desc("Integer out of range!");
       throw Bad_val<int>(i, desc);
}
```

As seen in the example above, `Opus_exceptions` types also can take a string description of the object (or some other useful information) accessible to the handler through the `str` member function. What happens in the throw statement is the constructor for `Bad_val<int>` is called with `i` and `desc` as arguments. The constructor for `Opus_exceptions` places a copy of the first argument in the public class member `arg` (in this case it would be of type `int`) where it is accessible to the exception handler, and stores the second argument to be used as a return value to the `str` member function. When no obvious type avails itself as the target of an exception, the template type `void*` with a value of 0 is commonly used.

Catching exceptions thrown by the library can be done in many ways depending on the desired result. However, be aware that *uncaught exceptions will terminate the application*. If the actual object that caused the exception is not important where it will be caught, no object to the catch statement need be specified:

```
try {
   something();
}
catch(Bad_val<string>) {
   cerr << "A bad value exception occurred." << endl;
}
```

If access to the exception object is desired, it should be caught as a reference:

```
try {
   something();
}
catch(Bad_val<string>& e) {
   cerr << "A bad value exception occurred." << endl;
   cerr << "Exception type: " << e.which() << endl;
   cerr << "Object causing exception: " << e.arg << endl;
   cerr << "Object description: " << e.str() << endl;
}
```

All `Opus_exceptions` types can be caught by a single catch clause of the form:

```
try {
   something();
}
catch(Opus_exceptions& e) {
   cerr << "Exception type: " << e.which() << endl;
   cerr << "Object description: " << e.str() << endl;
}
```

Opus_lock Class Hierarchy

The OAPI supports shared resource locking to ensure the integrity of items that might be subject to simultaneous change-access by more than one process. Common examples are OSF's and PSTAT's, but also include input/output files accessible by more than one process through the files blackboard. For example, it is important that a process intending to change an OSF be assured that no other processes will modify that OSF during the time it takes to perform its own change. The OAPI provides a means of obtaining an advisory lock[6] on a blackboard entry using the `lock` member function. ***It is up to the application developer, however, to obtain a lock on an entry before modifying it on the blackboard, and to release the lock once it is no longer needed. All pipeline applications must follow this protocol in order for locking to function well***.

The base class `Opus_lock` defines a generic interface for all resource locking in the OAPI. There are several types derived from `Opus_lock`:

- `Opus_lock_file` for locking files.
- `Null_lock` for cases where an `Opus_lock` object is needed, but no locking of the target object is required or meaningful (e.g., the `lock_entry` member functions on the time blackboards return locks of this type).
- `Osf_lock` and `Pstat_lock` are base classes for all OSF and PSTAT locks, respectively. They add a member function for retrieving the state of the OSF and PSTAT on the blackboard after the lock is acquired (the lock itself is a masked version of the actual PSTAT or OSF, so the lock does not contain this information).
- `File_osf_lock` and `File_pstat_lock` are file-based locks that inherit from `Opus_lock_file` and `Osf_lock` and `Pstat_lock`, respectively.

Utility Classes: Oresource & Pipeline

The `Oresource` class serves as an interface to resource files. Resource files, in general, contain keyword/value pairs and possibly, comment lines. Examples include process resource files, path files, pipeline stage files, etc. Keywords may contain a class-like

[6] Advisory locks require cooperation among all processes that might access a shared resource. In particular, a process must not alter a shared resource without first obtaining a lock. This is in contrast to *mandatory locks* that enforce locking at the access level and cannot be circumvented.

structure, and once the file is read into memory, additional keywords may be added to the in-memory copy.

The `Pipeline` class is a specialized version of `Oresource` (although it is not inherited from it) that is designed to parse pipeline stage files. This class makes it easier to extract pipeline definitions from these files.

Utility Classes: Msg

The `Msg` class offers an `ostream`-like interface to message reporting. Any number of `Msg` objects can be instantiated in an application; however, the data associated with these objects are static. That is, changes to one `Msg` object apply to all other `Msg` objects, and those changes remain in force even after the object goes out of scope (and is destroyed).

Utility Classes: Ofile, Opus_pid, Num_in_str

Several utility classes are part of the OAPI library. A brief summary of each appears in the following table:

Class	Purpose
`Ofile`	Can resolve VMS-like directory stretches in a file specification.
`Opus_pid`	Interface to the OPUS process ID that includes the host name and the system assigned parent process ID.
`Str_to_num`	Aids in conversion of string-embedded numbers to their native types.
`Num_in_str`	Aids encapsulation of numeric types in string objects.

Chapter 3

Application Development

This chapter covers the basics of building an application that uses the OAPI and important points to consider when developing your code. Chapter 4 examines three OAPI case studies in detail using the source code as a guide.

Memory Management

The OAPI creates and manipulates objects allocated off of the heap using `operator new` in most cases. Moreover, some member functions like `Entry::set_field` that act on their calling argument actually create a copy of the object passed to them (see **The Clone Idiom** in the next section) requiring that the caller dispose of the original version after the call. In general, it is the *client's responsibility* to free memory allocated for objects passed in this way to the OAPI and for those objects created in the OAPI and returned to the caller. Failing to do so will result in memory leaks. In particular, client code must guard against exceptions being thrown by the OAPI that might bypass delete operations on dynamically allocated objects.

The Clone Idiom

As mentioned in the previous chapters, the OAPI insulates clients from much of the library implementation through run-time polymorphism. The implication of this in C++ is the need to manipulate polymorphic objects through a pointer to the base type. Without resorting to run-time type identification, it is not possible in general to identify the actual object type referenced by the pointer. A commonly encountered situation both in OAPI and client code is one where a copy or duplicate of a polymorphic object is needed when only a pointer to the base type is held. To aid copying these objects, the `clone` virtual member function is included in many class interfaces. This member function creates an exact copy of the underlying object (off the heap; see **Memory Management** above) without the caller having to know the explicit object type referenced.

Unfortunately, not every C++ compiler supports covariant return types, so the `clone` member function always returns the new object as a pointer to the base type in the OAPI. A dynamic cast must be used on the newly constructed object, if necessary, to access member functions specific to that type's interface. For example, the OSF field type `Obs_stat` has the non-`const` member function `set_position` for changing the status character in the field for a particular processing stage. Given a `const` object, a clone must be made before calling a non-`const` member function:

```
const Obs_stat* ostat;
.
```

```
.
.
Field* copy = ostat->clone();
Obs_stat* ostat_copy = dynamic_cast<Obs_stat*>(copy);
ostat_copy->set_position(1, 'c');
.
.
.
delete ostat_copy;
```

Since the clone member function returns `Field*`, the new object must be downcast to the actual type before `set_position` is used as illustrated above.

Resource Locking

The OAPI supports shared resource locking to ensure the integrity of items that might be subject to simultaneous change-access by more than one process. Common examples are OSF's and PSTAT's, but also include input/output files accessible by more than one process through the files blackboard. For example, it is important that a process intending to change an OSF be assured that no other processes will modify that OSF during the time it takes to perform its own change. The OAPI provides a means of obtaining an advisory lock[7] on a blackboard entry using the `lock` member function. It is up to the application developer, however, to obtain a lock on an entry before modifying it on the blackboard, and to release the lock once it is no longer needed. All pipeline applications must follow this protocol in order for locking to function well. Locks are released by calling the `release` member function on a lock object or by deleting a dynamically allocated object.

The case studies in Chapter 4 illustrate use of this locking mechanism. Consider the following code fragment from the OSF event handling routine of Case #1:

```
// close event: lock event entries & call
// Opus_env::close_event
vector<Opus_lock*> locks;
evt->lock_list(locks);
// handle failed locks in a simplistic way
if (locks[0] == 0) {
   m << sev(Msg::W) << type(Msg::LOCK) <<
      "Failed to obtain lock on file - trying again..."
      << endm;
   sleep(5);
   evt->lock_list(locks);
   if (locks[0] == 0) {
      locks.clear();
      m << sev(Msg::E) << type(Msg::LOCK) <<
        "Failed to obtain lock again. No update " <<
        "will be performed." << endm;
}

// if no lock was obtained, ignore event
if (!locks.size()) opus_close_event(Opus_env::IGNORE_EVENT, evt);
else if (status)   opus.close_event("FILE_SUCCESS", evt);
else               opus.close_event("FILE_ERROR", evt);

if (locks.size()) delete locks[0]; // release lock
```

[7] Advisory locks require cooperation among all processes that might access a shared resource. In particular, a process must not alter a shared resource without first obtaining a lock. This is in contrast to *mandatory locks* that enforce locking at the access level and cannot be circumvented.

An attempt is made to lock each OSF entry in the event object using the `lock_list` member function prior to calling `Opus_env::close_event`. This is required because `close_event` will modify each of the event entries on the blackboard according to the supplied processing status[8] under the assumption that the caller already has locked each entry. Note that the client code must decide what to do in the situation where one or more locks could not be acquired. In the example above, a single retry attempt is made prior to giving up in this case. The locking mechanism itself makes several attempts to obtain a lock on an entry, so this example might be overkill. ***Nevertheless, it demonstrates that the advisory lock protocol used by the OAPI places responsibility for maintaining the integrity of shared resources squarely on the client.***

When an event occurs on a blackboard, a copy of the entries that triggered the event are placed in the event object. These entries are not locked on the blackboard at this point, so it is possible for these copies to diverge from their true blackboard states. (For example, if another process modified one or more of the entries after the copies were obtained.) If this happens and an attempt is made to update the entries listed in the event using `Opus_env::close_event`, an exception will be thrown. By locking the event entries using `lock_list` as demonstrated above, this issue is resolved automatically for OSF events—the locking mechanism refreshes each of the event entries with the blackboard state of the OSF after the lock is acquired. The same is not true for file events because parallel processing of files is not performed in general.

Compiling & Linking Against the OAPI

The OAPI has been built and tested using the following platforms and compilers:

- Compaq Tru64 UNIX v5.0A, Compaq C++ V6.2-024, Compaq C V6.1-013
  - Compiler flags: `-g -D__USE_STD_IOSTREAM -ptr [`*`template_repository_path`*`]`
- Sun Solaris 2.7, EGCS Release 1.1.2
  - Compiler flags: `-g`
- RedHat Linux 6.1, EGCS Release 1.1.2
  - Compiler flags: `-g`

Earlier versions of these operating systems and compilers may (or may not) work with the OAPI. The template repository path for Tru64 is the `obj/axp_unix/cxx_repository` sub-directory of the OPUS release. Header files are located under the `inc` sub-directory of the OPUS release and the OAPI library is located in the appropriate platform sub-directory of `lib` (`axp_unix` = Tru64; `sparc_solaris` = Solaris; `linux` = Linux).

Sample commands for each platform are given below for building the following OAPI code:

```
#include "msg.h"
#include "opus_env.h"
#include "event.h"
```

[8] Except when `Opus_env::IGNORE_EVENT` is used; see the documentation for `Opus_env::close_event`.

```
#include "halt_event.h"
#include "opus_exceptions.h"

using namespace std;

void halt_event_process(const string& title, Event* evt, const Opus_env& opus);

int main(int argc, char* argv[])
{
   Msg m;
   m.set_rpt_level(Msg::ALL);

   Opus_env opus(argc, argv);
   if (!opus.is_initialized()) {
      m << sev(Msg::F) << type(Msg::SEVERE) <<
        "OPUS failed to initialize." << endm;
      exit(EXIT_FAILURE);
   }

   Halt_event::add_callback(halt_event_process);

   try {
      while (true) {
         Event* e = opus.poll();
         e->process(opus);
         delete e;
      }
   }
   catch(Opus_exceptions& oe) {
      m << sev(Msg::F) << type(Msg::SEVERE) << oe.which() << endl;
      m << oe.str() << endm;
      return (EXIT_FAILURE);
   }
   catch(...) {
      m << sev(Msg::F) << type(Msg::SEVERE) <<
        "A non-OAPI exception occurred." << endm;
      return(EXIT_FAILURE);
   }
}

// Halt_event callback function
void halt_event_process(
                        const string& title,
                        Event* evt,
                        const Opus_env& opus)
{
   Msg m;
   m << "HALT command issued. Terminating." << endm;
   exit(EXIT_SUCCESS);
}
```

Assuming that OPUS v2.1A was installed in `/usr/local` and the above code was in a file named `sample.cpp` in the current working directory, the following commands could be used to create an executable named `sample` in the current working directory:

- Tru64

```
cxx -o sample -D__USE_STD_IOSTREAM -I/usr/local/opus/inc/ -
L/usr/local/opus/lib/axp_unix/ -ptr
/usr/local/opus/obj/axp_unix/cxx_repository/ sample.cpp -loapi
```

- Solaris

```
c++ -o sample -I/usr/local/opus/inc/ -
L/usr/local/opus/lib/sparc_solaris/ sample.cpp -loapi
```

- Linux

```
c++ -o sample –I/usr/local/opus/inc/ -L/usr/local/opus/lib/linux/
sample.cpp -loapi
```

Before running any application built against the OAPI, the environment variable `LD_LIBRARY_PATH` must be changed to include the path to `liboapi.so` (the OPUS installation and upgrade scripts configure `LD_LIBRARY_PATH` for you).

Chapter 4

Case Studies

A File Poller

The following example demonstrates a simple file polling application. A single file is polled for with the mask `*.pod`, then a set of operations are performed. Finally, the file is updated according to the status of the operations performed.

The process resource file and portions of the source code are presented below (a suitable path and pipeline stage file are not shown).

case1.resource:

```
PROCESS_NAME = case01
TASK = <oapi_sample_case1 -p $PATH_FILE -r case01>
DESCRIPTION = 'A Simple File Poller'
SYSTEM = 'OAPI Case Studies'
DISPLAY_ORDER = 1

FILE_RANK       = 1
FILE_DIRECTORY1 = inp_dir
FILE_OBJECT1    = *.pod

FILE_PROCESSING = _proc            ! set dangle to _proc

! new feature allows status to update arbitrary field (see Ch. 5)
FILE_SUCCESS.DIRECTORY = /done/    ! copy file to /done on success
FILE_SUCCESS.DANGLE    = _done     ! and change dangle to _done
FILE_ERROR.DIRECTORY   = /trouble/ ! copy file to /trouble
```

Trigger definitions include polling for a single file.

case1.cpp:

```
#include <iostream>
#include "opus_env.h"
#include "opus_lock.h"
#include "event.h"
#include "halt_event.h"
#include "file_event.h"
#include "msg.h"
#include "opus_exceptions.h"

using namespace std;

// event handler prototypes
void halt_event_process(const string&, Event*, const Opus_env&);
void file_event_process(const string&, Event*, const Opus_env&);

int main(int argc, char* argv[])
{
   Msg m;
   m.set_rpt_level(Msg::ALL);

   Opus_env opus(argc, argv);
   if (!opus.is_initialized()) {
```

Initialize OPUS.

```
      m << sev(Msg::F) << type(Msg::SEVERE) <<
        "OPUS failed to initialize." << endm;
      exit(EXIT_FAILURE);
   }


   Halt_event::add_callback(halt_event_process);
   File_event::add_callback(file_event_process);

   try {
      while (true) {
         Event* e = opus.poll();
         e->process(opus);
         delete e;
      }
   }
   catch(Opus_exceptions& oe) {
      m << sev(Msg::F) << type(Msg::SEVERE) << oe.which() << endl;
      m << oe.str() << endm;
      return (EXIT_FAILURE);
   }
   catch(...) {
      m << sev(Msg::F) << type(Msg::SEVERE) <<
        "A non-OAPI exception occurred." << endm;
      return(EXIT_FAILURE);
   }
}

// File_event callback function
void file_event_process(
                        const string& title,   /* Trigger name               */
                        Event* evt,            /* Pointer to the file event */
                        const Opus_env& opus)  /* Reference to Opus_env     */
{
   Msg m;

   Event::iterator vi = evt->begin();
   m << sev(Msg::D) << "Processing file: " <<
     (*vi)->str() << endm;

   // PROCESSING OF THE EVENT WOULD GO HERE
   int status = do_something();

   // close event: lock event entries & call
   // Opus_env::close_event
   vector<Opus_lock*> locks;
   evt->lock_list(locks);
   // handle failed locks in a simplistic way
   if (locks[0] == 0) {
      m << sev(Msg::W) << type(Msg::LOCK) <<
         "Failed to obtain lock on file - trying again..."
         << endm;
      sleep(5);
      evt->lock_list(locks);
      if (locks[0] == 0) {
         locks.clear();
         m << sev(Msg::E) << type(Msg::LOCK) <<
           "Failed to obtain lock again. No update " <<
           "will be performed." << endm;
   }

   // if no lock was obtained, ignore event
   if (!locks.size()) opus_close_event(Opus_env::IGNORE_EVENT, evt);
   else if (status)   opus.close_event("FILE_SUCCESS", evt);
   else               opus.close_event("FILE_ERROR", evt);

   if (locks.size()) delete locks[0]; // release lock
}

// Halt_event callback function
void halt_event_process(
```

Register callbacks.

Polling loop.

Locks must be obtained for the entries in the event before attempting to close the event.

```
                        const string& title,
                        Event* evt,
                        const Opus_env& opus)
{
   Msg m;
   m << "HALT command issued. Terminating." << endm;
   exit(EXIT_SUCCESS);
}
```

A Simple "Collector"

The following advanced example is a partial implementation of an application that polls for up to 10 OSF's with class "mem" that are waiting in the same "collection" stage. It then queries the database to determine whether any "associations" are complete. If all the members are present for an association, the association OSF (class "asn", assumed already to have been created) and its member OSF's are updated to show completion for the collection stage.

case2.resource:

```
PROCESS_NAME = case02
TASK = <oapi_sample_case2 -p $PATH_FILE -r case02>
DESCRIPTION = 'A Simple Collector'
SYSTEM = 'OAPI Case Studies'
DISPLAY_ORDER = 1

OSF_RANK               = 1
OSF_TRIGGER1.MAXTARGS  = 10
OSF_TRIGGER1.CO        = v
OSF_TRIGGER1.DATA_ID   = mem

OSF_PROCESSING.CO      = v  ! don't disturb OSF's after triggering

UPDATE_OSF.CO          = c  ! status used to update OSF's
```

Trigger definitions include polling for up to 10 OSF's of class 'mem' with 'v' in the stage CO

case2_main.cpp:

```
#include <iostream>
#include "opus_env.h"
#include "opus_lock.h"
#include "event.h"
#include "halt_event.h"
#include "osf_event.h"
#include "msg.h"
#include "opus_exceptions.h"

using namespace std;

// event handler prototypes
void halt_event_process(const string&, Event*, const Opus_env&);
void osf_event_process(const string&, Event*, const Opus_env&);

int main(int argc, char* argv[])
{
   Msg m;
   m.set_rpt_level(Msg::ALL);

   Opus_env opus(argc, argv);
   if (!opus.is_initialized()) {
      m << sev(Msg::F) << type(Msg::SEVERE) <<
        "OPUS failed to initialize." << endm;
      exit(EXIT_FAILURE);
   }
```

Initialize OPUS.

```
   Halt_event::add_callback(halt_event_process);
   Osf_event::add_callback(osf_event_process);

   try {
      while (true) {
         Event* e = opus.poll();
         e->process(opus);
         delete e;
      }
   }
   catch(Opus_exceptions& oe) {
      m << sev(Msg::F) << type(Msg::SEVERE) << oe.which() << endl;
      m << oe.str() << endm;
      return (EXIT_FAILURE);
   }
   catch(...) {
      m << sev(Msg::F) << type(Msg::SEVERE) <<
        "A non-OAPI exception occurred." << endm;
      return(EXIT_FAILURE);
   }
}
```

Register callbacks.

Polling loop.

case2_classes.h:

```
#ifndef CASE2_CLASSES_LOADED
#define CASE2_CLASSES_LOADED

#include <string>
#include <vector>
#include "opus_env.h"
#include "event.h"

class Association {
public:
    Association(const string&);     // construct given association name

   void add_member(const string&); // add member to this association

                                   // remove members from Event
   void purge_evt_members(const Opus_env&, Event*) const;

   string name() const;            // get association name

private:
   const string name;              // association name
   vector<string> members;         // list of association members
};

#endif
```

case2_classes.cpp:

```
#include "case2_classes.h"
#include "osf.h"
#include "dataset.h"

using namespace std;

Association::Association(const string& aname) : name(aname) {}

void Association::add_member(const string& m)
{
   // don't allow duplicates
   if (members.find(m) == members.end()) members.push_back(m);
}

void Association::purge_evt_members(const Opus_env& opus, Event* e) const
{
   // create search template
```

```
   Osf* osf = opus.new_osf();
   osf->search_mask_all();
   Dataset* dset = new Dataset;

   // remove members from Event (if present)
   typedef vector<string>::const_iterator VI;
   for (VI i = members.begin(); i != members.end(); i++) {
      dset->assign(*i);
      osf->set_field(dset);
      try {
         e->remove_entry(osf);
      }
      catch(...) {
         // ignore no match
      }
   }
   delete osf;
   delete dset;
}

string Association::name() const
{
   return(name);
}
```

case2_evt_hdlrs.cpp:

```
#include <iostream>
#include <vector>
#include "opus_env.h"
#include "opus_lock.h"
#include "event.h"
#include "msg.h"
#include "obs_stat.h"
#include "pstat.h"
#include "osf.h"
#include "opus_exceptions.h"
#include "case2_classes.h"

using namespace std;

// Osf_event callback function
void osf_event_process(
                       const string& title,   /* Trigger name              */
                       Event* evt,            /* Pointer to the OSF Event */
                       const Opus_env& opus)  /* Reference to Opus_env    */
{
   Msg m;

   m << sev(Msg::D) << "Processing OSF(s): " << endl;
   Event::const_iterator vi;
   for (vi = evt->begin(); vi != evt->end(); vi++) {
      m << (*vi)->str() << endl;
   }
   m << endm;

   vector<string> asscns;
   vector<Association*> alist;
   try {
      // get list of possible associations
      get_all_asscns(evt, asscns);

      // check if any of those associations are
      // completed by the event entries
      prune_incompletes(asscns, evt, alist);
   }
   catch(Severe<void*>) {
      m << sev(Msg::E) << type(Msg::SEVERE) <<
         "Failed to obtain database information for event processing; "
         "ignoring event." << endm;
      opus.close_event(Opus_env::IGNORE_EVENT, evt);
```

Call functions to process the event.

```
      return;
   }

   // attempt to locate & update association ID for
   // each complete association
   update_ass_osf(opus, evt, alist);
   for (int i = 0; i < alist.size(); i++) delete alist[i];

   // mark members complete
   vector<Opus_lock*> locks;
   evt->lock_list(locks);

   // handle failed locks in a simplistic way
   int i = 0;
   vi = evt->begin;
   while(vi != evt->end) {
      if (locks[i] == 0) {
         m << sev(Msg::W) << type(Msg::LOCK) <<
            "Failed to obtain lock on OSF - trying again..."
            << endm;
         sleep(5);
         try {
            locks[i] = evt->lock_entry(*vi);
         }
         catch(...) {
            // mark item as failed and issue error
            m << sev(Msg::E) << type(Msg::LOCK) <<
              "Failed to obtain lock again. Dropping entry:"
              << endl << (*vi)->str() << endm;
            failed_entries.push_back((*vi)->clone());
         }
      }
      vi++;
      i++;
   }
   // remove failed entries from event
   for (i = 0; i < failed_entries.size(); i++) {
      evt->remove_entry(failed_entries[i]);
      delete failed_entries[i];
   }

   // close event
   opus.close_event("UPDATE_OSF", evt);

   for(int i = 0; i < locks.size(); i++) delete locks[i];

}

 // Halt_event callback function
void halt_event_process(
                        const string& title,
                        Event* evt,
                        const Opus_env& opus)
{
   Msg m;
   m << "HALT command issued. Terminating." << endm;
   exit(EXIT_SUCCESS);
}
```

Locks must be obtained for the entries in the event before attempting to close the event.

case2_process.cpp:

```
#include <vector>
#include <string>
#include "event.h"
#include "dataset.h"
#include "case2_classes.h"

using namespace std;

void get_all_asscns(
                    const Event* e,         // I - OSF event
```

```
                    vector<string>& asscns) // O - list of possible associations
{
   Msg m;

   // for each member, get association it belongs to
   Event::const_iterator ei = e->begin();
   Dataset* d = new Dataset;
   string s;
   while(ei != e->end()) {
      (*ei)->get_field(d);                 // fetch dataset name from OSF
      try {
         DB_get_assn_for_mem(d->ustr(), s); // gets assn. name given member name
      }
      catch(...) {
         m << sev(Msg::E) << type(Msg::MISSING) <<
           "Association name lookup failed for: " <<
           d->ustr() << endm;
         delete d;
         throw Severe<void*>(0);
      }
      asscns.push_back(s);                 // place association name in vector
      m << sev(Msg::D) << d->ustr() " belongs to " << s << endm;
      ei++;
   }
   // remove duplicates
   sort(asscns.begin(), asscns.end());
   vector<string>::iterator vi = unique(asscns.begin(), asscns.end());
   asscns.erase(vi, asscns.end());

   delete d;
}

void prune_incompletes(
                       vector<string>& asscns,      // I/O - possible assns
                       Event* e,                    // I   - OSF event
                       vector<Association*>& alist) // O - complete associations
{
   Msg m;

   // for each association, check if all members present
   vector<string>::iterator vi = asscns->begin();
   Event::iterator ei = e->begin();
   Osf* mosf  = dynamic_cast<Osf*>((*ei)->clone()); // will need an OSF later
   Dataset* d = new Dataset;
   string s;
   vector<string> mems;
   vector<Entry*> mem_its;
   bool incomplete;
   Association* ac;
   while(vi != vi->end()) { // loop over each possible association
      try {
         DB_get_assn_members(*vi, mems); // gets member names for assn.
      }
      catch(...) {
         m << sev(Msg::E) << type(Msg::MISSING) <<
           "Association member lookup failed for: " <<
           *vi << endm;
         delete d;
         delete mosf;
         throw Severe<void*>(0);
      }
      incomplete = false;
      for(int i = 0; i < mems.size(); i++) {
         d.assign(mems[i]);
         mosf->search_fill_all();  // create search template
         mosf->set_field(d);
         if ((ei = e->find_entry(mosf)) == e->end()) {
            incomplete = true;
            break;
         } else {
            mem_its.push_back((*ei)->clone());
```

```
         }
      }
      if (incomplete) { // remove any existing members from event
         for(int i = 0; i < mem_its.size(); i++) {
            e->remove_entry(mem_its[i]);
            delete mem_its[i];
         }
         m << sev(Msg::D) << "Association incomplete: " << *vi << endm;
      } else {
         m << sev(Msg::D) << "Association complete: " << *vi << endm;

         // create association object
         ac = new Association(*vi);
         for(int i = 0; i < mem_its.size(); i++) {
            mem_its[i]->get_field(d);
            ac->add_member(d->ustr());
            delete mem_its[i];
         }
         alist.push_back(ac);
      }
      mem_its.clear();
      mems.clear();
      vi++;
   }
   delete d;
   delete mosf;
}

void update_ass_osf(
                    const Opus_env& opus,              // I - Opus_env object
                    Event* evt,                        // I - the Event
                    const vector<Association*>& alist) // I - associations
{
   // create search template
   Osf* aosf = opus.new_osf();
   aosf->search_fill_all();
   Data_id* asscls = new Data_id("asn");
   aosf->set_field(asscls);
   delete asscls;

   Dataset*  assnm = new Dataset;
   vector<Entry*> res;
   typedef vector<Association*>::const_iterator VI;
   for (VI i = alist.begin(); i != alist.end(); i++) {
      assnm.assign((*i)->name());
      aosf->set_field(assnm);  // set Dataset name to search on
      res.clear();
      if (opus.find_osf(aosf, res)) {
         Osf_lock* lock = 0;
         try {
            lock = opus.lock_osf(res[0]); // assumes unique OSF
            delete res[0];
         }
         catch(...) {
            Msg m;
            m << sev(Msg::E) << type(Msg::LOCK) <<
               "Failed to acquire lock on " << (*i)->name() <<
               ". Skipping ASN." << endm;
            (*i)->purge_evt_members(evt); // remove members from Event
            delete res[0];
            continue;
         }
         Osf* osf = lock->get_target();
         opus.apply_status(osf, "UPDATE_OSF");
         delete osf;
         delete lock; // release lock & free memory
      } else {
         Msg m;
         m << sev(Msg::E) << type(Msg::MISSING) <<
           "Could not locate ASN OSF: " << (*i)->name() << endm;
         (*i)->purge_evt_members(evt); // remove members from Event
```

```
            }
        }
        delete aosf;
        delete assnm;
    }
```

Non-Pipeline Application

The following example demonstrates how a non-pipeline application gains access to the OPUS blackboards and utilities through `Opus_env`. The code is identical to the actual tool `osf_update`.

`osf_update.cpp:`

```
#include <iostream>
#include <stdlib.h>
#include <string>
#include <vector>
#include "opus_env.h"
#include "field.h"
#include "osf.h"
#include "dataset.h"
#include "data_id.h"
#include "dcf_num.h"
#include "obs_stat.h"
#include "msg.h"
#include "sys_public.h"

using namespace std;

const char usage[] = "Usage:\n\t osf_update -p pathfilename -f filename"
                     " [-t type] \n\t\t [-n number] [-m new_dcf] [-c column]"
                     " -s status\n\nExample:\n\t osf_update -p blue.path -f "
                     "n32s1496 \n\t\t -t nic -n 123 -c DP -s p \n\n"
                     "note: the path file must be in OPUS_DEFINITIONS_DIR \n\n";


int main(int argc, char* argv[])
{
    Msg m;

    //
    // initialize as non-pipeline app
    //
    Opus_env opus(argc, argv, false);
    if (!opus.is_initialized()) {
       m << sev(Msg::E) << type(Msg::SEVERE) <<
          "osf_update - Failed to initialize OPUS environment." << endm;
      exit(EXIT_FAILURE);
    }

    //
    // validate command-line args
    //
    string path;
    string obs_name;
    string status;
    vector<string> values;
    opus.get_option(string("-p"), val
    if (values.size() == 0) {
      m << sev(Msg::E) << type(Msg::MISSING) <<
          "osf_update - Missing required argument: -p" << endl << usage << endm;
       exit(EXIT_FAILURE);
    }
```

The third argument to the `Opus_env` constructor indicates that this is a non-pipeline application.

Command-line parsing is up to the application in this case.

```
path = values[0];
values.clear();
opus.get_option(string("-f"), values);
if (values.size() == 0) {
   m << sev(Msg::E) << type(Msg::MISSING) <<
      "osf_update - Missing required argument: -f" << endl << usage << endm;
   exit(EXIT_FAILURE);
}
obs_name = values[0];
values.clear();
opus.get_option(string("-s"), values);
if (values.size() == 0) {
   m << sev(Msg::E) << type(Msg::MISSING) <<
      "osf_update - Missing required argument: -s" << endl << usage << endm;
   exit(EXIT_FAILURE);
}
status = values[0];
values.clear();

//
// initialize path
//
char pnm[FILENAME_MAX];
SYS_void_get_file_name(path.c_str(), pnm);
opus.set_path(string(pnm));
```

The `set_path` member function instantiates the OSF and PSTAT blackboards.

```
//
// create new OSF object in memory
//
Osf* osf = opus.new_osf();
osf->search_fill_all();

//
// set individual fields
//
Field* f;
f = new Dataset(obs_name);
osf->set_field(f);
delete f;

opus.get_option(string("-t"), values);
if (values.size() > 0) {
   f = new Data_id(values[0]);
   values.clear();
   osf->set_field(f);
   delete f;
}

opus.get_option(string("-n"), values);
if (values.size() > 0) {
   f = new Dcf_num(values[0]);
   values.clear();
   osf->set_field(f);
   delete f;
}

//
// search for OSF
//
vector<Entry*> results;
int num;
if (num = opus.find_osf(osf, results)) {
   if (num > 1) {
      m << sev(Msg::E) << type(Msg::AMBIG) <<
         "osf_update - More than one (count=" << num << ") OSF matched: "
        << endl << osf->str() << endl << "Matches:" << endl;
      for (int i = 0; i < num; i++) m << (results[i])->str() << endl;
      m << endm;
      exit(EXIT_FAILURE);
   }
   //
```

```
    // obtain OSF lock
    //
    Osf_lock* lock;
    delete osf;
    if (!(osf = dynamic_cast<Osf*>(results[0]))) {
       m << sev(Msg::E) << type(Msg::SEVERE) <<
         "osf_update - Unable to cast search result to OSF." << endm;
       exit(EXIT_FAILURE);
    }
    try {
       lock = opus.lock_osf(osf);
    }
    catch(...) {
       m << sev(Msg::E) << type(Msg::SEVERE) <<
         "osf_update - Failed to obtain lock for OSF." << endm;
       exit(EXIT_FAILURE);
    }

    //
    // apply DCF NUMBER update
    //
    Osf* new_osf = dynamic_cast<Osf*>(osf->clone());
    opus.get_option(string("-m"), values);
    if (values.size() > 0) {
       f = new Dcf_num(values[0]);
       values.clear();
       new_osf->set_field(f);
       delete f;
    }

    //
    // apply status update
    //
    Obs_stat* ostat;
    vector<string> columns;
    opus.get_option(string("-c"), columns);
    if (columns.size() == 0) {
       ostat = new Obs_stat(status);
    } else {
       ostat = new Obs_stat();
       osf->get_field(ostat);
       opus.get_option(string("-s"), values);
       if (columns.size() != values.size()) {
          m << sev(Msg::E) << type(Msg::MISUSE) <<
            "osf_update - The number of -c options does not match the "
            "number of -s options." << endm;
          delete lock;
          exit(EXIT_FAILURE);
       }
       int pos;
       for (int i = 0; i < columns.size(); i++) {
          pos = opus.get_stage_position(columns[i]);
          ostat->set_position(pos, (values[i])[0]);
       }
    }
    new_osf->set_field(ostat);
    delete ostat;

    //
    // apply change to blackboard
    //
    opus.replace_osf(osf, new_osf);

 } else {
    m << sev(Msg::E) << type(Msg::MISSING) <<
      "osf_update - Unable to locate OSF: " << endl << osf->str() << endm;
    delete lock;
    exit(EXIT_FAILURE);
 }

 //
```

```
    // Release lock
    //
    delete lock;

    return (EXIT_SUCCESS);
}
```