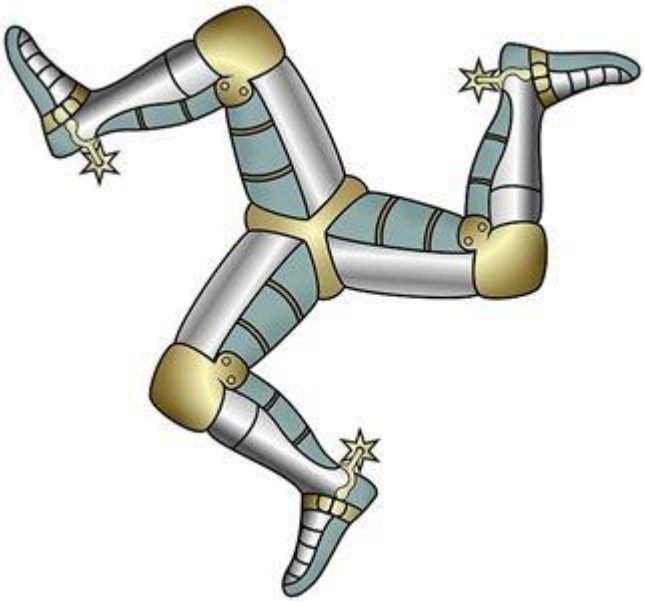
DASHING PLATFORMS

DEVELOPER'S HANDBOOK



1.0 Coding Standards

This purpose of this section is to outline the necessary elements for consistency in producing our software's source code. Consistency is the most important factor in creating a development environment that fosters understanding and growth. This document is meant to address the following languages: C, C++, and Java.

1.0.1 Common File Extensions

For the C language, source files should be named with a '.c' extension, and header files should be named with a '.h' extension.

For the C++ language, source files should be named with a '.cpp' extension, and header files should be named with a '.h' extension.

For the Java language, source files should be named with a '.java' extension.

Other common file extensions will be as follows: Data files will end in a '.dat' extension. Configuration files will end in a '.conf' extension. Text files will end in a '.txt' extension, and Script files will end in a '.sh' extension regardless of the shell interpreter used.

1.0.2 Source Templates

Template files are available for the various source languages that we use. These templates outline the structure of our source code. Template SHOULD be used for each new development file created. The location of the template files will be discussed in the following section on the development / loadbuild environment.

1.0.3 Typographic Restrictions

Each source file line should be restricted to within 80 to 120 characters. This allows us to be able to print (portrait or landscape) THE most important design artifact—the code itself.

1.0.4 Source Files

Source files should be logically named to provide descriptive information about their

individual purposes.

1.0.4.1 Limited File Scope

The scope of a single source file should be limited to the purpose for which it was created. This improves readability and maintenance. For C++ and Java, there should be one class per file. Any other class should be a nested class within the primary class.

1.0.4.2 Grouping of Source Files

Groups of classes should be logically grouped into directories or packages based on their functionality. This guideline applies to all of the programming languages; the development / loadbuild environment discussed in the following sections has been designed with packaging in mind.

NOTE: When file scope and grouping evolve to a point that the original organization is lost or no longer makes sense, refactor that piece of code immediately—do not let disorganization or feature creep get out of control.

1.0.4.3 Naming Techniques for Files and Directories

For directories, the entire directory name should remain lower case. This must be part of the consideration when deciding on package names in Java and on modules names in CORBA IDLs.

For files, we capitalize the first character of each word in the name. This aids in delimiting the word boundaries. For example, a source file name could be MessageDataArray.cpp, and it could be located in the package directory /platform/messagestruts/

1.0.5 Function, Class, Macro, and Variable Naming

Class names should generally match that of the filename; that is, capitalize the first character of each word in the name. This also applies to Structure and typedef enum variables. For example, class MessageDataArray and typedef struct MessageData MessageDataType (Note the use of the 'Type' identifier at the end of the name—this is used to indicate passive behavior which is the case with traditional structs).

Function and Method names should begin with a lower case character, but the beginning of each subsequent word should be capitalized. For example, void processMessageData(void).

Variable names should follow the same convention as Function and Method names—they should begin with a lower case character, but the beginning of each subsequent word in the name should be capitalized. Additionally, member variables of a class in C++ and Java should end with a '_' character. For example 'memberVariableData_'. Rather, local, stack, and parameter variables should NOT end in '_'.

Macros, variable defines, static member constants, and global constants should be all CAPITALIZED. Use the underscore, '_', character to replace spaces between words and special characters. For example, you might declare the following: const int MESSAGE_POOL_SIZE=5;

1.0.6 Descriptive Names and Abbreviations

Treat standard abbreviations like words. For example, you might use variable names such as 'ipAddress' or 'corbaDataStructure'. The purpose of variable names is to be as descriptive as possible. This improves readability and maintenance of our software components. Non-standard abbreviations should be avoided. For example, we should use variable names such as 'basicOriginationRequest' instead of 'borreq'.

1.0.7 Accessor and Mutator Methods in C++ and Java

For accessor and mutator methods, use 'get' and 'set' prefixes to denote usage. For example, use 'getMessageData()' for accessors and 'setMessageData()' for mutators to clearly indicate the intent of the method.

1.0.8 Include and Import Statements

Within a C/C++ header file, restrict how many other files are included to only those files that are relevant to the header file itself. Where possible, includes should be deferred to the source file instead. Remember that a header file's sole purpose is to provide definitions and expose interfaces. It should not be used as a convenient place to include files needed by source files.

Inlining source code in C/C++ header files (implementation residing in header files,

rather than a corresponding .c/.cpp file) is STRICTLY PROHIBITED. Inlining code for performance optimization reasons is not sufficient; optimization will be performed at a system level as product maturity evolves. Inlining source code in header files breaks package independence and prevents a consistent patching strategy.

Each C/C++ header file should contain an Include Guard to prevent it from being included in a library module or binary executable more than once:

```
#ifdef _MESSAGE_HANDLER_CLASS_
#define _MESSAGE_HANDLER_CLASS_
<body of the header file>
#endif
```

1.0.9 Exceptions

Do not use exceptions to handle normal events. Exceptions should only be used if something unexpected occurs. The overhead involved in throwing exceptions is significant enough that it should be avoided for routine events.

Do not throw exceptions inside constructors. In C++, this can cause a memory leak if not handled correctly. In general, it is inadvisable to throw exceptions. Instead, consider adding an initialize() or activate() method to perform the code that can generate the exception.

1.0.10 Assertions

Avoid the use of assertion statements in C++.

1.0.11 Unique Module Identification

A single source file per module should be chosen to contain the module-level identification string. We use a compile define, BUILD_LABEL, passed-in from the make command line (which must contain NO spaces) for setting the identifier:

```
/* From the C++ FAQ, create a module-level identification string using a
compile define, BUILD_LABEL, which must contain NO spaces, passed
in from the make command line */
#define StrConvert(x) #x
#define XstrConvert(x) StrConvert(x)
static volatile char mainSCCSId[] __attribute__ ((unused)) = “@(#)App”
```

```
"\n Build Label: " XstrConvert(BUILD_LABEL)
"\n Compile Time:  " __DATE__ " " __TIME__;
```

where 'App' should be replaced with the module name. Note that the above "@(#)" string should be exact—it is used in SCCS identification using standard UNIX utilities.

1.0.12 Control Statements

If a control statement can have a nested block, then it must define it. For example, a 'for' loop must contain a nested block even if it has only one statement. For example:

```
for(int i = 0; i << 10; i++)
{
  counter++;
}
```

The above braces must be used to declare the block.

1.0.13 Comment Formatting

Comments that begin a line should have the first word capitalized and be punctuated like a regular sentence. Comments that appear at the end of a line containing source code should be lower case and omit end-of-sentence punctuation.

Doxygen is integrated into the development / loadbuild environment in order to generate javadoc-like design documentation of the source code. Since Doxygen can recognize multiple styles of source code comments, refer to the Doxygen documentation, and try to maintain consistency in our code commenting style.

1.0.14 Indenting and Spacing

We must all strive to follow a standard indenting and spacing mechanism. As presented in the Template source files, the following indentation and spacing rules should be observed:

- Standard indent is 3 spaces
- Opening braces should begin on a new line (not K&R style)
- Use the common file header sections from the Template source files
- Use an itemized parameter list
- Follow the structure provided for defining a class in a header file
- Same format for functions in header and source files

NOTE: TAB characters are not allowed. Each developer is responsible for configuring their editor to replace TABs with spaces (3 spaces). TABs will not be tolerated in the source code!!

1.0.15 Specific Guidelines for C++

1.0.15.1 Virtual Destructors

Any class that has or might eventually have sub-classes must employ a virtual destructor in order to avoid a memory leak. Unless you are completely certain that a class will never be sub-classed, you are almost always best served by making every destructor virtual. Certain objects that get created often and possess a short lifespan may still be chosen not to do so.

1.0.15.2 Catch Exceptions by reference

All exceptions should be caught by reference:

```
try{...}
catch (SystemException& se) {...}
```

Note the previous warning about the appropriate use of exceptions.

1.0.15.3 Always Use an Initializer List in Constructors

For initializing the internal data members of a class, an initializer list in each and every constructor should be used. This promotes efficiency because the compiler will initialize internal data objects anyway. For this reason, delaying any initialization until the body of the constructor represents a source of inefficiency and therefore should be avoided. Note that an initializer list is also the ONLY place from which you can call the base class constructor. This is an important language feature.

1.0.15.4 Do Not Rely on the Compiler's Automatic Functions

In C++, it is impossible to define a class with no member functions. The compiler will automatically generate the following operations for you:

- Default Constructor
- Copy Constructor
- Default Destructor

- Assignment Operator (operator=)
- Address-Of Operator (operator&)

Relying on the compiler to generate these methods automatically for a class represents a potential danger area where errors can be introduced. The default implementation may not provide the exact behavior needed.

Often, extraneous copying can occur in a system simply because the developer relied on, or forgot, that the compiler would provide these. This is difficult to diagnose since nothing appears in the source code to indicate this type of mistake.

In another situation, a class that provides one or more non-default constructors (meaning that they take parameters) is intended not to have a default (parameter-less) constructor used. However, the presence of these non-default constructors will effectively "hide" an absent default constructor. If the intention is not to have the default constructor called accidently, it should be declared in the private section of the class.

In general, unused default constructors, copy constructors, and assignment operators should be declared private so that they cannot be accidently called. There are occasions when you do not want clients of a particular class to be able to instantiate or copy an object at will. This will force application designers to consciously make a choice.

2.0 Development and Loadbuild Environment

The development and loadbuild environment has been built upon Jacob Dreyer's Makefile System (from Geotechnical Software Services – Geosoft). This Makefile system has been constructed to allow C, C++, and Java components to be built and packaged together in a consistent manner. Usage of the Makefile system is provided under the terms of the GNU GPL license.

Full documentation on how the Makefile System works can be found in the development environment directory tree at the following location:

dev/make/javamake.html

Please read this documentation to understand how the Makefile system works, as it is somewhat different in how it is structured from other Makefile systems.

The development environment is organized into the following files and directories:

- loadbuild.txt – Step by step, in sequence instructions for building the source
- setEnv.sh – Contains environment variables that must be sourced prior to attempting any work with the Makefile system. This file must be customized to match the locations of third party components. For deployment, this file should be copied to /etc/platformInit.conf as this is where the init.d script will look for it.
- todo.txt – Log file of tasks planned and currently underway
- bin – Binary executables and supporting shell scripts and configuration files for invocation
- docs – Doxygen generated source code comments created by invoking 'make doxygen'
- lib – Dynamic (shared) libraries and Static (archive) libraries automatically created by the Makefile system during compilation and linking
- make – Support files for the Makefile system. Additionally, this directory contains Template Source Code files for each of the supported languages.
- misc – Contains miscellaneous files for investigative purposes, sample or example code files, and reference documents
- src – Source code tree. All product source code is stored here.
- obj – Mirror directory of the 'src' directory, but contains object files created during compilation.

The Makefile system supports the following make targets:

- make BuildTag=some_buildtag_without_spaces_010101
- make clean
- make idl (for initial creation of stubs and skeletons from IDL files—this only needs to be re-run when IDL changes occur)
- make doxygen (for creation of source code comment documentation)
- make statistics (for count of lines of code)
- make depend (for makedepend dependency tree documentation)

3.0 External Libraries and Documentation

This section contains descriptions of the various third-party libraries used within the product:

ACE – The Adaptive Communication Environment (ACE) is an open source C++ network programming API, created by Doug Schmidt and supported by several university research groups, that includes support for some additional functions and utilities. Assuming that ACE is installed the standard directory, the following links should be reviewed to understand the capabilities of ACE:

/opt/ACE_wrappers/docs/index.html
/opt/ACE_wrappers/html/index.html

Sample code that uses ACE can be found in the following locations:

/opt/ACE_wrappers/examples

/opt/ACE_wrappers/tests

And, the sometimes the most detailed understanding of ACE comes from its own source code:

/opt/ACE_wrappers/ace/*

The following 3 helpful books on ACE development are available from Amazon and other technical bookstores:

C++ Network Programming, Vol 1: Mastering Complexity with ACE and Patterns by Douglas C. Schmidt, Stephen D. Huston

C++ Network Programming, Vol 2: Systematic Reuse wth ACE and Frameworks by Douglas C. Schmidt, Stephen D. Huston

The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming by Stephen D. Huston

TAO – The ACE Orb is a CORBA 2.3/2.4 compliant orb that is built atop ACE. It too was created by Doug Schmidt and is supported by several university groups (commercial support is additionally available from several companies). The TAO root installation directory is located inside the ACE root directory. The following links should be reviewed to understand the capabilities of TAO:

/opt/ACE_wrappers/TAO/docs/index.html

Sample code that uses TAO can be found in the following locations:

/opt/ACE_wrappers/TAO/examples

/opt/ACE_wrappers/TAO/tests

Additionally, information about the TAO IDL compiler or the CORBA Orb Service components, examine the following locations:

/opt/ACE_wrappers/TAO/TAO_IDL

/opt/ACE_wrappers/TAO/orbsvcs

And of course, additional understanding can be obtained by examining the TAO source code in the following directories:

/opt/ACE_wrappers/TAO/tao

/opt/ACE_wrappers/TAO/orbsvcs/orbsvcs

While the following book is not specific to the TAO Orb, the syntax and use of commands and libraries closely compares with TAO. The TAO developers recommend this book to their users:

Advanced CORBA Programming with C++ by Henning and Vinoski

JacOrb – An Open source CORBA 2.3/2.4 compliant Orb developed in Java by a German University research group (commercial support available). The following can be examined for further information on JacOrb:

/opt/JacORB_2_2_1/index.html
/opt/JacORB_2_2_1/doc/ProgrammingGuide.pdf

GCC / G++ - With the exception of the Java components, all portions of the product are compiled with the GNU GCC/G++ compiler. Development has been conducted on either GCC 3.2.3 or 3.4.3. Refer to the following web address for documentation on the compiler:

http://gcc.gnu.org

Refer to the following directories for header file documentation of the C++ libraries (including STL) that are available in the compiler:

/usr/include/c++/3.2.3 -or-
/usr/include/c++/3.4.3

PostgreSQL – Open source relational database. Here we use the Pervasive Postgres distribution. Pervasive offers commercial support for the Postgres database. We currently use the libpq client side 'c' library for interfacing to the database (as well as the database administration and shell utilities). The complete documentation for Postgres and all of the required utilities is installed by the Pervasive installation program in the following location:

/opt/pervasivepostgres/docs/html/index.html

3.1 Platform Components

In this section, the architecture, design, and usage of each of the platform components is discussed. As mentioned previously, the product source code is located in the /dev/src directory. NOTE: Complete source code documentation can be browsed in the /dev/docs directory after using 'make doxygen' to invoke the loadbuild process. Below is a list of the src directory hierarchy:

The various ems packages in the 'ems' Project directory:

/ems/cliclient
/ems/clientagent
/ems/guiclient
/ems/idls
/ems/idls/cppsource
/ems/idls/session
/ems/idls/platformConfig

The various platform packages in the 'platform' Project directory:

/platform/common
/platform/datamgr
/platform/logger
/platform/msgmgr
/platform/opm
/platform/processmgr
/platform/resourcemon
/platform/utilities

The various sql script files for creating the product schema and populating default datafill in the 'sql' Project directory:

/sql/

The various unit test and utility packages in the 'unittest' Project directory:

/unittest/cleanloggerSM
/unittest/datamgrtest
/unittest/loggertest
/unittest/loggertest2
/unittest/msgmgrgrouptest1
/unittest/msgmgrgrouptest2
/unittest/msgmgrtest1
/unittest/msgmgrtest2
/unittest/msgmgrtest2sm
/unittest/msgmgrtest3
/unittest/msgmgrtest3sm
/unittest/opmtest
/unittest/test1
/unittest/test2
/unittest/versionid

NOTE: The README file in the unittest directory contains the unittest inventory with descriptions; it must be updated with any additions to that directory.

3.1.1 Logger

The logger subsystem components provide a trace logging facility for the product.

Logging functionality is broken into two pieces:

- Logger client interface
- Log Processor

The following sections will describe these in detail. Refer to the unittest directory packages for examples on how to use the Logger.

3.1.1.1 Logger Client Interface

The application developer is responsible for initializing the Logger Client by including the "Logger.h" header file, using the static singleton instance method to create the interface, and calling the initialization method:

```
#include "platform/logger/Logger.h"
Logger* logger = Logger::getInstance();
logger->initialize(true);
```

NOTE: the boolean value "true" passed to the initialize method tells the client to send log output stdout. This is useful for debugging the application without being dependent on the other platform components (such as the Log Processor). Passing a boolean "false" value will cause the Logger client to enqueue the log message in a shared memory queue for the Log Processor to dequeue and process.

The application developer has 3 methods available for generating logs. Each of the following is a C-style Macro that serves to hide the complexity of the logging interface. By using these Macros for logging, we can potentially change the logging interface for the product later without impacting all of the existing source code. Furthermore, these Macros allow us to add the source code filename and line number (and other fields) for each log without the developer having to specify them.

- TRACELOG(level, subsystem, msg, arg1, arg2, arg3, arg4, arg5, arg6)
- TRACELOGLITE(level, subsystem, msg, arg1, arg2, arg3, arg4, arg5, arg6)
- STRACELOG(level, subsystem, msg)

where:

- 'level' is enumerated to be: ERRORLOG, WARNINGLOG, INFOLOG, DEBUGLOG, or DEVELOPERLOG (defined in "platform/common/Defines.h")
- 'subsystem' is a typedef enum given for each subsystem that generates logs (defined in "platform/common/Defines.h" with string form defined in "platform/logger/LoggerCommon.cpp"
- 'msg' a char* pointing to the log message. In the case for TRACELOG and TRACELOGLITE, this log message should be specified using printf string formatting characters

- arg variables are for printf string formatting character substitutions

Note that STRACELOG assumes that the log message has lots of dynamically built information and that string formatting has already been performed. The difference between TRACELOG and TRACELOGLITE is that the LITE version does not invoke getpid inside the macro to stamp the log with its originating process id. 'getpid' is the only system call performed inside the Logger Client, so the LITE Macro can be used to eliminate this overhead in very highly demanding sections of the code. VERY IMPORTANT: transient (stack allocated) string (char*) variables must not be passed as parameters to TRACELOG / TRACELOGLITE as they will be deallocated when the stack goes out of scope while they are still in the shared memory log queue (this will cause a core dump).

While debugging application modules, the developer may manually control the subsystem log levels. Normally these will be controlled through the EMS. Note the following syntax to enable Developer Logs for My New Application:

```
#include “platform/logger/Logger.h”
Logger* logger = Logger::getInstance();
logger->setSubsystemLogLevel(MY_APPLICATION, DEVELOPERLOG);
```

If shared memory logging is enabled (by passing 'false' to Logger initialize), then setting the Subsystem log level will set the corresponding value in shared memory (Logger Config) for all Logger Clients running on the same node to use—this is how the platform updates the log levels for all processes when a change is requested from the EMS client. It will also update the relational database with the same value for persistence. If shared memory logging is disabled for debugging, then setting the subsystem log level will only affect the current process.

3.1.1.2 Log Processor

The Log Processor is invoked as a child process of the ProcessManager. The Log Processor performs dequeuing of LogMessages from shared memory and sends them to the specified output mechanism. The following output mechanisms are supported (via a command line option passed in when the P
ProcessManager invokes the Log Processor):

-f <filename> Send output to the specified logfile
-o Send output to the OS Syslog facility
-s Send output to stderr / stdout. This is Default.

When the '-f' option is specified to enable platform management of log files, the

following additional options may be specified:

-n <number> Number of log files to preserve for rollover
-z <size> Size of each log file to allow prior to rollover

Note that when we are doing our own log file management (-f mode option specified), then:

a.) if '-n' option is specified, then log files will rollover where the current active log file always remains <filename>, the next oldest log file will be <filename>.1, and the oldest log file will be <filename>.(n - 1) -- after which, the log files will be deleted from the filesystem. Note that this style of rollover is consistent with how the syslog daemon does it.
b.) If '-n' is specified as well as the '-z' option, then each log file will be limited to <size>. If '-n' is specified, but '-z' is omitted, then a default size will be used for each log file.
c.) If '-n' is omitted, then log files will NOT rollover, and the specified <filename> will grow until manually cleaned up (recommended for developers).
Also note that when the syslog option is selected, stdout / stderr will not be redirected because of a system limitation.

3.1.2 OPM (Object Pool Manager)

The OPM provides memory pool management functionality for C++ objects on a per process basis. Application developers design their C++ source code to inherit from the OPM Base classes (for the objects that they wish to pool / cache). Then, they request OPM to create a Pool of these objects. During application execution, the objects are continually requested from these many pools to do work and then released back into the OPM pools when their work is complete.

OPM allows the developer to control each object pool's growth behavior individually. Pool side may be statically fixed, allowed to grow (if the need arises), or allowed to grow and shrink (using a controllable hysteresis algorithm). The pool manager supports tracking object ownership internally—transparent to the developer. The OPM also keeps usage statistics for allowing pool sizes to be individually tuned for system profiling and performance optimization.

The OPM exists as a shared library which can be initialized and invoked by each of the application processes. The OPM itself is NOT a process started by the Process Manager. Refer to the unittest directory packages for examples on how to use the OPM, and refer

to the doxygen documentation for interface specific details.

3.1.3 Message Manager and Timer Framework

The MsgMgr framework provides an abstraction ontop of the ACE network programming mechanisms. The MsgMgr creates a facade of using 'Mailboxes' to communicate between applications and cards as well as within a single application process. Any system component that needs to communicate with another system component will create a Mailbox and register it with the Mailbox Lookup Service, which is responsible for proxy routing and delivery of Messages from one Mailbox to another. When an application needs to send a message to another component, it asks the Mailbox Lookup Service to find the other component's mailbox based on the supplied well-known address, then it posts the message to the mailbox.

The MsgMgr framework handles almost all serialization and deserialization semantics for the developer. The developer uses a callback method registration mechanism so that automatic handlers for the various message types will process all of the received messages. This mechanism is the basis for all Application Concurrency control.

Additionally, the MsgMgr framework encapsulates the ability to create application layer single-use and periodic Timers as well as perform timer management functions such as canceling and restarting timers.

Routing of Mailbox Messages is accomplished using a self-described Mailbox Address. The Mailbox Address is a complex structure that can describe the following characteristics of an Application's Mailbox:

- Mailbox Name – A recognizable string for representing the mailbox (maybe the same as the Application's Name)
- Physical or Logical Type – Physical addresses are typically associated with the geographical slot or location in the system (For example, card is slot 3). Logical addresses are associated with a role that the network element is playing (For example, active card in an active/standby redundant pair).
- ShelfNumber – Geographical shelf address
- SlotNumber – Geographical slot address in the shelf
- Location Type – Specifies the transport mechanism used by the Mailbox. This is determined by the type of communication that the Mailbox is involved in. The following are supported: Local Mailbox, Local Shared Memory Mailbox, Distributed Mailbox, Group Mailbox

- InetAddress – IP Address used in distributed and group mailbox communcations
- Redundancy Role – Preferred role of the network element when configured in a redundant manner, such as: active, standby, or loadshared.

The Mailbox facade serves to standardize how messages are exchanged and handled throughout the product; the transport layer technology can be varied with virtually no change to the source code. The application developer is responsible for determining their communication needs so as to choose the appropriate transport layer mechanism. The following are some general rules for Mailbox Location Types:

- Local Mailboxes are used for multi-threaded communications within the same process. This mailbox type provides for the highest performance, as no buffer copies and no serialization/deserialization are performed. Local Mailboxes encapsulate the Timer functionality of the MsgMgr framework.
- Local Shared Memory Mailboxes are used to post and receive messages between threads in different processes of the same card/computer. This mailbox type has performance overhead due to buffer copies into and out of shared memory.
- Distributed Mailboxes are used to post and receive messages between applications on different cards/computers. This mailbox type has the greatest performance cost due to the message serialization and deserialization that must be performed.
- Group Mailboxes are used to post and receive messages simultaneously between multiple applications on different cards/computers. This mailbox type uses the Reliable Multicast Protocol and serializes / deserializes messages to multicast groups.

Note that the last 3 mailbox types have an associated Local Mailbox with them. This is so an application that needs both Local and non-Local (out of process) communications will transparently have a single Mailbox that performs both functions. (An example would be an application that needs to send and receive messages from an application on another card, but it also needs to schedule Timers through a Local Mailbox).

Like the OPM, the MsgMgr framework exists as a shared library which can be initialized and invoked by each of the application processes. The MsgMgr itself is NOT a process started by the Process Manager. Refer to the unittest directory packages for examples on how to use the MsgMgr framework, and refer to the doxygen documentation for interface specific details.

3.1.4 Data Manager

DataManager provides an application interface for setting and retrieving data to and from the persistent store. DataManager is provided to hide some of the complexity of working with the database. It is intended to allow the database to be transparently replaced by different products/mechanisms without requiring the applications to do major surgery. Additionally, the DataManager defines the concurrency model for how the applications will interact with the database. For example, an application program will have multiple threads; each thread requires periodic access to the database. A connection set will be created for this application that contains FEWER connections than the number of threads that need them. Each thread is required to reserve a database connection, use it, and then release it back into the DataManager pool so that another thread can then reserve and access. Of course the number of pooled connections versus the number of application threads is a systems engineering exercise and is configurable. Note that each connection is itself NOT thread safe and multiple threads should not attempt to access the same connection object (DbConnectionHandle) after it has been reserved by a single thread (this is typically the concurrency architecture for most database drivers). (This is possible with additional mutex/concurrency mechanisms implemented by the application, but it is NOT ADVISABLE, and not necessary). Note that the object given back to the application by Data Manager is a DbConnectionHandle; this is a wrapper object intended to protect the real connection object from misuse by the developer.

The general error handling strategy for the application developer should be as follows. If any errors are encountered while issuing query/update/insert/delete commands or while analyzing results, the developer should release the connection back into the pool and attempt to reserve another connection for use. The release process includes a health check on the connection that will attempt to re-establish/renew that connection in the event of a failure.

DataManager exists as a library for each of the application processes to use for managing its database connections. DataManager supports pooling of connections, and connection re-establishment and retry. Because the DataManager is a bootstrap requirement for the applications obtaining all of their other configuration information (from the database), the DataManager reads ITS configuration from a file. Furthermore, because DataManager is used by multiple processes, this file must contain process-specific configuration sections to meet each process's unique database connection

requirements (with regards to the number of connections, both local and remote, etc.)

The format of the DataManager configuration file is:

```
[Configuration]
DEBUGLEVEL=5

[ConnectionSet Name]
# Some comments
# Perhaps some more comments
USERID=userid
PASSWORD=password (NOTE: This will need to be encrypted
        later...suggest MD5)
DRIVERNAME=libpq
CONNECTIONTYPE=LOCAL (or REMOTE)
CONNECTIONSTRING="hostaddr=127.0.0.1 port=3180"
NUMBERCONNECTIONS=1
DSN=someDSN

[Next ConnectionSet Name]
...
```

The above [Configuration] section exists in order to set DataManager-wide configuration parameters. Each subsequent [ConnectionSet Name] section establishes the configuration for a pool of connections. Note that an OPM object pool is created even if only 1 connection is indicated. An application process can have multiple [ConnectionSet] sections for its use in the configuration file. The application process would then call DataManager::activateConnectionSet(string connectionSetName) for each configured connection set that it needs to use.

The required steps for using the Data Manager can be seen in the unit test 'dataManagerTest', and should be performed as follows (note that proper error handling has been omitted below). First, initialize the Data Manager with the location of the Data Manager configuration file:

```
DataManager::initialize("./DataManager.conf");
```

Then the application should activate one or multiple connection sets using the header name for that connection set in the configuration file. For example:

```
DataManager::activateConnectionSet("LoggerConnection");
```

When the application is ready to perform some database operation, it should reserve a connection from the pool. In the event that there are currently no free connections in the connection set pool, then this method call will block until a connection becomes available (by another portion of the application 'releasing' a connection). If the application cannot afford to block while waiting for a connection, it can alternatively call 'reserveConnectionNonBlocking' which may return a NULL DbConnectionHandle. See the following:

```
DbConnectionHandle* connectionHandle =
        DataManager::reserveConnection("LoggerConnection");
```

If the application is using multiple Data Manager connections, and it needs to know which operations were performed on each connection object, the application can set a user-defined identifier on each connection handle object. This identifier will be preserved across reserve/release cycles.

```
string tmpString("someIdentifier");
connectionHandle->addUserDefinedIdentifier(tmpString);
```

At any time, the application can check the health of the database:

```
connectionHandle->checkConnectionHealth();
```

Query, insert, update, and delete operations can be performed by the application:

```
connectionHandle->executeCommand("select * from loglevels");
```

The result sets of query operations can be parsed with the following methods:

```
int columnCount = connectionHandle->getColumnCount();
int rowCount = connectionHandle->getRowCount();
//string columnName = connectionHandle->getColumnName(column);
for (int row = 0; row < rowCount; row++)
{
   for (int column = 0; column < columnCount; column++)
   {
      cout << connectionHandle->getValueAt (row, column) << endl;
   }//end for
}//end for
```

When all result set processing is complete, the application should clean up the result set to prevent a memory leak:

```
connectionHandle->closeCommandResult();
```

And when all operations have been completed, the application should release the connection back into the connection set pool:

```
DataManager::releaseConnection(connectionHandle);
```

Note that all of the above API methods has been designed to be very generic so that the underlying relational database driver can be swapped out with another database product's driver in the event of a vendor change. In such a case, there should be no impact to the applications or to the Data Manager interfaces.

SQL scripts used for creating the database schema as well as populating the default datafill are located in /dev/src/sql.

The Data Manager exists as a shared library which can be initialized and invoked by each of the application processes. The Data Manager itself is NOT a process started by the Process Manager. Refer to the unittest directory packages for examples on how to use the Data Manager, and refer to the doxygen documentation for interface specific details.

3.1.5 Process Manager

The Process Manager provides for application process startup, shutdown, suspend, resume and monitoring. The Process Manager itself should be invoked by the Linux init service using the /etc/init.d/platformInit Linux service script.

Process Manager uses the ACE Service Configurator framework combined with ACE's own Process Management libraries. As part of this, Process Manager reads from an ACE configuration file which normally defaults to svc.conf—we have renamed it to /dev/bin/ProcessManager.conf. By changing this file to indicate that a particular application should be shutdown or restarted -and- by issuing a SIGHUP signal to the Process Manager, each application process can be individually controlled. This is directly applicable to our maintenance and patch management strategies.

Process Manager accepts the following startup options:

```
-b Daemonize the Process Manager process (not needed and should not
        be used as the daemonization is performed in the source code)
-d Verbose diagnostics as process directives are forked
```

-f Read a different file other than svc.conf
-n Do not process static directives
-s Designate which signal to use for reconfigure instead of SIGHUP
-S Pass in a process directive not specified in the svc.conf file
-y Process static directives
-l Local Logger Mode only (do not enqueue logs to shared memory)
-r Redirect stdout/stderr to a specified file (to be removed...)

The ProcessManager.conf configuration file that drives the Process Manager accepts the following format. For statically linked services (these are optional features that will run within the Process Manager OS process):

static <service name> ["argc/argv options for the service"]

For dynamically linked services, for example, those services intended to run within their own OS process:

dynamic <service name> <service type> <DLL/.so name>:<factory function> ["argc/argv options"]

where the 'service type' is either a Service_Object *, Module *, or Stream* -and- where the "argc/argv options" have the following format:

"<process executable name> <'NULL' or redirected IO filename> <process options>"

The following additional service directives are supported:

remove <service name>	shuts down the service/process
suspend <service name>	suspends activity for the process
resume <service name>	resumes activity for the process

The process manager provides for the graceful shutdown of application processes. Each application process is required to implement the signal handler and shutdown handler as specified in the /dev/make/TemplateApp.cpp template file. NOTE that catchShutdownSignal is static and calls the instance 'shutdown()' method. Application specific shutdown routines should be added to 'shutdown()'.

3.1.5 Fault Manager

3.1.6 Resource Monitor

The Resource Monitor is invoked as a child process of the Process Manager. This platform application provides for auditing of vital card/computer resources including cpu, memory, and disk usage. Historical trend data is stored for these resources and overload control thresholds are monitored to alarm unhealthy conditions. The Resource Monitor gathers much of its information directly from the Linux /proc filesystem.

The Resource Monitor accepts the following command line options:

-l Local Logger Mode only (do not enqueue logs to shared memory)

3.1.6 CORBA Naming Service

The product relies on the Common Object Brokerage Request Architecture (CORBA) for communication between the server components and the management client, as well as for external interfaces to customer side systems. Our product is compliant with at least version 2.3 of the Object Management Group (OMG) CORBA Standard. We have chosen to implement the C++ CORBA components using the open source TAO Orb, a CORBA implementation which is itself implemented using the ACE framework. The Java CORBA components are implemented using JacOrb, an open source Java Orb, which is well known for its OMG CORBA Standards compliance and its interworking with TAO.

The CORBA Naming Service is a distributed component that allows for the registration of network-visible objects using a unique, but well-known, name. There existed 3 different Naming Service products that were immediately available for us to use:

TAO Naming Service
JacOrb Naming Service
Sun's J2SDK Corba Naming Service

Based on its close compliance with the OMG CORBA 2.3 Standard, in particular its support for the Interoperable Naming Service (INS) addendum to the standard, we have chosen to use the TAO Naming Service.

The TAO Naming Service accepts the following command line arguments:

-o <file path location to which to output the Naming Service root IOR>
-f <memory mapped file path location to use to persist object references>
-p </var/run filename to output the Naming Service pid to>
-ORBListenEndpoints 'iiop://<ip>:<port>' url for determining which interface to use on a multi-homed host (localhost IP address is allowable)
-ORBDottedDecimalAddresses 1 for specifying that IP addresses should

be used inside IORs instead of hostnames in order to prevent an uneeded reliance on DNS resolution.

The Naming Service should be started prior to the other CORBA components in the system. It can either be started by the Process Manager, or it can be started by the Linux init service and allowed to daemonize.

3.1.7 EMS Client Agent

The EMS Client Agent is the server component that accepts GUI and CLI client connections for managing and maintaining the system. The Client Agent is invoked as a child process of the Process Manager, and it accepts the following command line parameters:

-l Local Logger Mode only (do not enqueue logs to shared memory)
-h (use IP addresses instead of hostnames in IORs)
-d (enable Orb debug logs)
-v <debug level> (Max is 10)
-f <log filename> (Default is stdout)
-e <Endpoint> (Use the endpoint URL for which interfaces to listen for connections on)
-i <Initial Reference> (URL for initial resolution of the Name Service)

Note that the correct syntax for -i (the ORBInitRef) -ORBInitRef is:

NameService=corbaloc:iiop:localhost:12345/NameService
-- or --
NameService=iiop://localhost:12345/NameService

The former (corbaloc) syntax is the standard (compliant with the Interoperable Naming Service specification) syntax. The latter syntax is TAO-specific and is deprecated.

Upon startup, the Client Agent:
a.) uses the corbaloc interoperable naming service Url to determine the initial (bootstrap) reference of the Naming Service. The Client Agent then binds its top level object references into the Naming Service using a unique, but well known, name. Next, the Client Agent attempts to resolve the remote object references for any clients that have already registered with the Naming Service.
b.) initializes its connection to the database using the Data Manager. It then reads the Log Level configuration information and sets it in shared memory for all of the processes to use. (If the database connection cannot be established, then log level configuration defaults to what is specified in the log library). After initialization, Client Agent will

respond to CLI/GUI commands to change log levels (and other configuration data) by updating it in the database and in shared memory if applicable (all done through the platformConfig idl interface).

Once initialized, the Client Agent maintains two thread groups: one for processing messages received from the other system components via its Mailbox and the MsgMgr framework, and one for processing messages received from its remote clients via the CORBA Orb. Messages from the Client Agent to either the Clients or another MsgMgr mailbox can originate from either thread group.

3.1.8 EMS CLI Client

The EMS CLI Client is a menu-driven, command line client for allowing users access to the EMS Client Agent. The CLI Client is intended to be run on the host card/computer that the product runs on; it should be automatically invoked by the shell upon login of the CLI users. The CLI Client accepts the following command line parameters:

-h (use IP addresses instead of hostnames in IORs)
-d (enable Orb debug logs)
-l <debug level> (Max is 10)
-f <log filename> (Default is stdout)
-e <Endpoint> (Use the endpoint URL for which interfaces to listen for connections on)
-i <Initial Reference> (URL for initial resolution of the Name Service)

Upon start, the CLI Client uses the corbaloc interoperable naming service Url to determine the initial (boostrap) reference to the Naming Service. The CLI Client then binds its top level object references into the Naming Service using a unique, but well known, name. Next, the CLI Client attempts to resolve the remote object reference for the Client Agent from the Naming Service.

Once initialized, the CLI Client maintains two thread groups: one for processing user initiated menu selections, and one for processing messages received from the Client Agent via the CORBA Orb.

The CLI Client has menu options to support the following:

1. Heartbeat between Client and Agent
2. Displaying all subsystem log levels
3. Setting a specific subsystem log level

3.1.9 EMS GUI Client

The EMS GUI Client is a Java application client for allowing users access to the EMS Client Agent. The GUI Client is intended to be run on a remote host computer. The GUI Client accepts the following command line parameters:

-ORBInitRef NameService='corbaloc::<ip>:<port>/NameService

as well as the necessary classpath parameter.

3.1.10 Unit Tests

Unit tests should be developed for all functionality. Additional unit tests should be developed for source code resulting only in library modules (rather than binary executables, which can aid in the testing effort).

An inventory of unit test modules is kept in /dev/src/unittest/README