Date: October 20, 2004

Subject: ACE/OSAL Comparison Report
Project: Multi-Mission System Architectural Platform (MSAP)

Introduction:

This report provides an inspection comparison (i.e.: without performance data) of two different operating system (OS) abstraction layer software products. One is the Goddard Space Flight Center (GSFC) developed Operating System Abstraction Layer (OSAL), and the other is the Washington University, St. Louis developed Adaptive Communication Environment (ACE).

The purpose of these abstraction layers is to support software development that is independent of the host operating system. It should be mentioned there are other aspects of an operating environment that hinder host independent development such as hardware specific devices and board level support. This support extends to include physical characteristics of the underlying hardware over which the software must operate. This report focuses on the operating system aspects of an abstraction layer. In other words, those layers releasing the developer from working with the likely vagaries of a specific operating system. For reference, there are hardware device level abstractions available such as the Abstract Device Model (ADM) developed for X2000, and to a lesser extent portions of the Goddard developed OSAL[1] referenced in this report.

The operating system facilities such as task management, semaphores, and inter-process communication are emphasized in this report. As abstraction layers, both ACE and OSAL provide APIs that support application development without making direct calls to the underlying operating system. In order to achieve the benefits (such as portability) from using an operating system abstraction layer, it is extremely important to require all higher-level development use the abstractions and not develop to the underlying operating system calls.

This report provides a brief overview of each product followed by a more detailed comparison that focuses on the differences between ACE and OSAL implementations. As stated earlier, this report is a visual comparison of the products. If desired and with additional time, performance measurements can be collected and reported. One possible approach for these measurements is to retool the VML component evaluation (task completed last month) from using ACE to OSAL.

Lastly, the ease of using either product cannot be completely understood or evaluated without spending sufficient development time working with each product. At the end, a brief conclusion and opinion is provided.

---

[1] The Goddard OSAL library contains both hardware and operating system abstraction layers. This report focuses on the operating system portions of the OSAL library. A portion of the hardware abstraction dealing with processor specific type information (e.g.: 16-bit integer) is mentioned in this report.

ACE Introduction:

ACE is a freely available, object-oriented framework that implements many core design patterns for multi-threaded software applications. Originally developed as a university based research and development effort, ACE is implemented in C++ and provides support for many operating system platforms. These platforms include both Wind River VxWorks and Sun Solaris (supporting both POSIX and Solaris thread models), both of which are required platforms for MSAP. Additional platforms include Mac OS X, Linux (Unix/POSIX in general), and Microsoft Windows.

ACE is comprised of four functional layers. Namely, these layers are the OS adapter, C++ wrapper facades, frameworks, and distributed services. The primary goal for ACE is to support common middleware services and reusable design patters that are largely independent of OS (and language to some extent). Accomplishing this goal requires abstracting the operating system from intermediate layers. This report focuses on the OS adapter layer that resides directly above the native APIs of the target operating system and provides the closest comparison (call-wise) to the GSFC OSAL implementation.

The ACE OS adapter layer encapsulates APIs for multi-threading, thread synchronization, inter-process communication, shared memory, event distribution, etc. In short, it focuses on providing the multi-thread support functions of the abstraction layer.

For compilation, ACE makes use of conditional compilation. The lowest levels of ACE rely on preprocessor defined variables to determine which portions of the software to compile. This method minimizes much of the common code duplication across source files, but can often leads to untidy source files as more conditional variables are added. Also, the ACE code contains frequently nested conditional compilation statements that can become eye straining. The following ACE code fragment shows a simple example of nested variables. This example demonstrates a nested VxWorks definition within an ACE_HAS_THREADS definition.

```
int
ACE_OS::cond_destroy(ACE_cond_t *cv)
{
  ACE_OS_TRACE ("ACE_OS::cond_destroy");
# if defined (ACE_HAS_THREADS)
#   if defined (ACE_HAS_WTHREADS)
  ACE_OS::event_destroy (&cv->waiters_done_);
#   elif defined (VXWORKS) || defined (ACE_PSOS)
  ACE_OS::sema_destroy (&cv->waiters_done_);
#   endif /* VXWORKS */
  ACE_OS::thread_mutex_destroy (&cv->waiters_lock_);
  return ACE_OS::sema_destroy (&cv->sema_);
# else
  ACE_UNUSED_ARG (cv);
  ACE_NOTSUP_RETURN (-1);
# endif /* ACE_HAS_THREADS */
}
```

OSAL Introduction:

OSAL is available as a GSFC open source software product downloadable and controlled under the terms of the NASA Open Source Agreement (NOSA). A copy of the NOSA is available from https://opensource.gfsc.nasa.gov/nosa.php.

OSAL is a thin library, developed in C, which isolates embedded software from the underlying real-time operating system. It provides an abstraction layer to both the hardware and the host operating system. The primary goal of OSAL is to allow the development of portable system software that is independent of a particular real-time operating system.

OSAL provides an implementation for Wind River VxWorks. However, there is no documented support for a Solaris workstation environment. The complete list of supported operating systems include RTEMS, VxWorks, Mac OS X, and Linux. The Linux implementation uses POSIX, so it may be relatively simple to derive Solaris support from their Linux baseline. The OSAL source code tree is comprised of branches, or subdirectories, for each supported operating system. A pre-build configuration file is tailored to specify to desired operating system target. When compiled, the directories (see Figure 1) containing the specific operating system source code files are compiled.

The OSAL source code tree separates common code from hardware architecture and operating system specific code. Figure 1 - OSAL Source Code Tree contains a diagram of the OSAL build directory structure. All operating system specific code is maintained within a separate directory for each supported operating system. There are no pre processor conditional compilation directives used. Root level configuration files are used to determine which portions of the directory structure to incorporate into a library build.

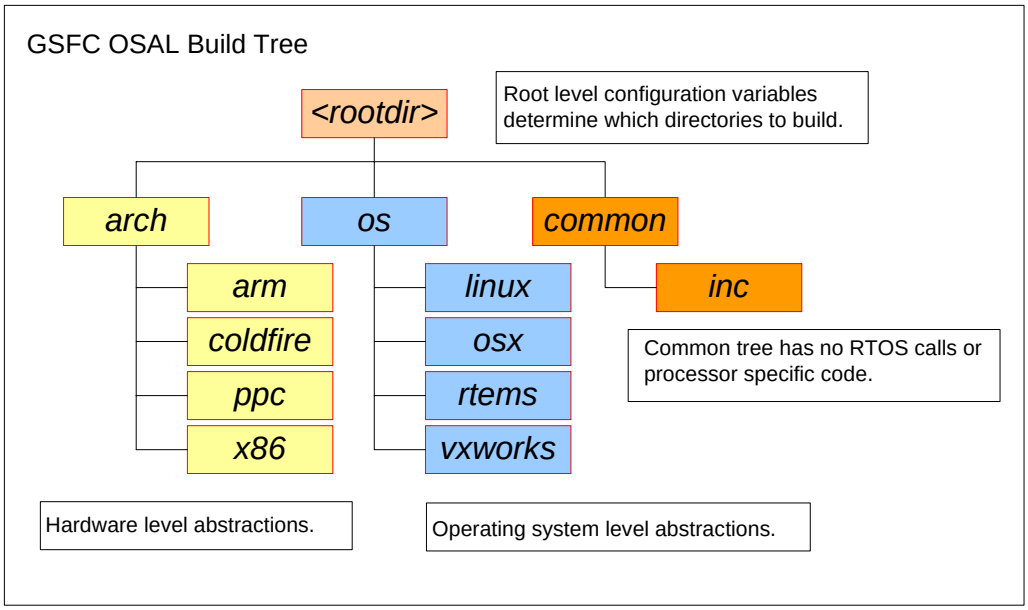

Figure 1 - OSAL Source Code Tree

Porting software to support additional platforms is handled by extending the directory structure to accommodate the new target. The benefit to this layout is that code is completely isolated for each specific hardware and software target. The drawback is that there can be a significant amount of code duplication within each of the target directories. Any source code file that contains both common and OS specific code is maintained in its own directory.

The Comparison:

The comparison emphasizes the differences between ACE and OSAL focusing on the areas of task control, message queues, and thread synchronization. These three areas represent the majority of OSAL. As mentioned earlier, ACE provides higher levels of abstraction (middleware services and design patterns), but these are only briefly mentioned in this report since there are no OSAL equivalents.

Task Control:

The OSAL task control APIs are implemented as a very thin layer above the operating system specific task control calls. A compile time parameter is used to encode the maximum number of allowed threads. This parameter is checked each time a new thread is created with an error returned when the maximum number of new threads is exceeded.

After checking the maximum number of allowed threads, OSAL invokes the native operating system thread spawn call before returning a status. Appendix A contains a code listing of the OSAL VxWorks thread create function. One thing to notice is that the calling parameters map directly (by meaning and type) to the native operating system call.

By comparison, the ACE APIs provide multiple layers of abstraction and also introduce abstraction into the calling parameters. In some cases, the calling parameters are other ACE objects. Using ACE objects as parameters adds a degree of type safety into the programming. For example, ids for thread identification can be typed objects with specific behavior and used only as intended and not mixed with other parameters. The programmer cannot pass/use a thread id as a semaphore id for example.

One other thing to notice is the OSAL call passes the thread priority directly to the underlying operating system call. One aspect I was not able to verify is how ACE interprets thread priorities. In any case, I believe operating system attributes whose values result in different behavior across different operating systems must be part of the abstraction. For example, the abstraction should define the range of supported priorities and runtime calculate the equivalent priority for each supported operating system.

Both OSAL and ACE provide calls to manage thread behavior once they have been created. This includes functions such as thread delay, change priority, etc.

Message Queues: *(Comments in this area are still being compiled. ACE/OSAL Queue Create, Get, Put).*

Thread Synchronization:

In much the same way OSAL supports threads, OSAL provides a single level of abstraction supporting both semaphores and mutexes. A compile time parameter is used to define the maximum number of allowed semaphores and mutexes. At runtime, this

parameter is checked and an error is returned if the maximum number of semaphores and mutexes is exceeded. Appendix B4 contains a listing the OSAL supported semaphore and mutex API calls. For its VxWorks implementation, OSAL uses the mutual exclusion semaphore library (semMLib) and the binary semaphore library (semBLib). There is no support for counting or shared memory type semaphores.

By comparison, ACE provides a set of lock classes that provide thread synchronization objects. Portions of these classes offer mutex, thread mutex, and process mutex support providing a simple and efficient mechanism that serializes access to shared resources (such as a file or object in shared memory). These methods encapsulate native Solaris, and POSIX operating system calls. Other lock classes offer semaphore, thread semaphore, and process semaphore support providing a general mechanism for serializing multiple threads of control. ACE supports counting and condition variable semaphores, as well as recursive and non-recursive locks.

One feature I found very helpful was ACE Guards. When developing, it is easy for programmers to forget to make mutex unlock calls. This can starve other threads that are trying to acquire a lock. Furthermore, deadlocks will occur if the owner tries to reacquire a lock it already owns. The ACE mutex abstraction, called Guards, ensures locks are automatically released using the semantics of C++ constructors and destructors. The code placement of the Guard, at the beginning of a critical section, will invoke a constructor acquiring the initial lock. Equally, the destructor of a Guard automatically releases the lock when the Guard goes out of scope. This model improves code correctness by automating the initialization of locks using C++ classes. They also guarantee that locks are released automatically.

Conclusion:

In conclusion, the differences between the OSAL and ACE build environments are usually not a tradeoff factor since both code baselines are externally maintained. The differences start becoming important once a project decides to maintain either of these code baselines internal to the project. Using OSAL would potentially require changes to support a Solaris workstation environment.

This report focused on thread control, message queues, and thread synchronization. This represents a very small fraction and a different objective for ACE and what ACE provides. ACE supports a variety of data structures and functional categories. These categories include containers, configuration, message queues, connectors, and others.

The primary goals for ACE and OSAL are very different. ACE was developed to address middleware services and design patterns independent of operating system and development language (but still object oriented). Maybe not intentional, but a consequence of this goal was the ACE OS abstraction layer. On the other hand, OSAL was developed to provide a lightweight single level of coding abstraction over the native operating system.

A selection decision involves tradeoffs. For example, one may welcome the simple and very lightweight interface OSAL offers. On the other hand, one may decide to accept more complicated interfaces in return for additional value added functionality, and safeguards. You also cannot ignore implementation language differences. Most programmers perform development in a single language. This is not to say one language is better than another, but code mixing C functions with C++ objects can be distracting.

From my analysis, OSAL does not perform any dynamic memory allocation. On the other hand, there are portions of the ACE layer that perform runtime memory allocation. However, I have not examined if these memory allocations are associated with the lowest ACE OS abstraction level.

In my opinion, the benefits ACE offers are best gained in a C++ programming environment that can take advantage of the object-oriented programming techniques and safeguards to help develop better code. This advantage comes in terms of reliability, maintainability, etc. There is very little benefit in writing C wrappers around a handful of ACE classes. This would result in a simpler calling interface much like what OSAL already provides, but it would also leave overhead from several untapped ACE features.

In my opinion, there are currently not enough benefits to using OSAL as a product library. For us, a better approach might be to develop our own abstraction layer using OSAL (and code from other JPL missions such as MER) for some initial ideas. Within a few weeks, we would be in a much better position than trying to incorporate OSAL into our development requirements.

One last thing to mention, both ACE and OSAL provide type definitions for hardware specific data sizes. For example, an unsigned integer is typed as uint32 in OSAL. We also

need to define these types in our *TBD* abstraction layer. In addition to the operating system abstraction layer, there is other common software functional areas to consider such as logging, debug tracing, and code instrumentation. These functional areas represent frequently used code that is called by higher-level modules.

My recommendation is to include an additional analysis of the MER OS abstractions and make a follow-up recommendation for MSAP.

Points of Contact:

OSAL
Alan Cudmore
NASA Goddard Space Flight Center
Code 582.0
Greenbelt, MD 20771
Alan.P.Cudmore@nasa.gov

ACE
University of Washington
Sat Louis, MO. 63130-4899
http://www.cs.wustl.edu/~schmidt/ACE.html

Appendix A: OSAL Task Create Function

```
uint32 OS_TaskCreate (uint16  task_id,
                      char   *task_name,
                      void   *function_pointer,
                      uint32 *stack_pointer,
                      uint32  stack_size,
                      uint32  priority,
                      uint32  flags)
{

      /* Check Parameters */
      if( (task_id > OS_MAX_TASKS) ||
            (OS_task_table[task_id] != 0) )
      {
            return OS_ERROR;
      }

      /* Create VxWorks Task */
      OS_task_table[task_id] = taskSpawn(task_name, priority,
            flags, stack_size, (FUNCPTR)function_pointer,
            0,0,0,0,0,0,0,0,0,0);

      if(OS_task_table[task_id] == ERROR)
      {
            return OS_ERROR;
      }

      return OS_SUCCESS;
}
```

Appendix B: OSAL API List

B1. Miscellaneous APIs:

- OS_InfoGetResetType
- OS_InfoGetProcessorId
- OS_InfoGetTaskID
- OS_InfoGetTicks
- OS_InfoGetEnvironment
- OS_InfoGetNetworkID

B2. Task Control APIs:

- OS_TaskCreate
- OS_TaskDelay
- OS_TaskSetPriority
- OS_TaskRegister
- OS_IncExecCount
- OS_GetExecCount

B3. Queue APIs:

- OS_QueueCreate
- OS_QueueGet
- OS_QueuePut

B4. Semaphore and Mutex APIs:

- OS_BinSemCreate
- OS_BinSemGive
- OS_BinSemTake
- OS_BinSemTimedWait
- OS_MutSemCreate
- OS_MutSemGive
- OS_MutSemTake