Version: 11/26/2002 9:22 PM

FARGOS/VISTA

Object Management Environment

Programmer's Reference

Release 4.1.2

FARGOS/VISTA Object Management Environment Programmer's Reference Manual

FARGOS Development, LLC
757 Delano Road
Yorktown Heights, NY 10598
http://www.fargos.net
mailto:support@fargos.net

Abbreviations

FARGOS Development, LLC is a Limited Liability Company registered with the State of New York. It is required to identify itself as such in its name, hence the ", LLC" suffix. For purposes of readability in this document, the ", LLC" suffix is sometimes dropped. The phrase "FARGOS Development" always denotes "FARGOS Development, LLC" and is not intended to suggest any alternate form of organization.

Contents

1. Introduction

The FARGOS/VISTA Object Management Environment provides the infrastructure of a transparently distributed, multithreaded, object-oriented operating system that operates across heterogeneous systems. FARGOS/VISTA-based applications are realized as active objects that reside within and interact with this distributed environment.

The multithreaded nature of the FARGOS/VISTA Object Management Environment makes it ideal for implementing asynchronous applications. The true power of FARGOS/VISTA is realized by making use of its distributed capabilities, which provide a robust, powerful and easy-to-use foundation for building complex, peer-to-peer distributed applications that work across a variety of hardware and operating system platforms.

The FARGOS/VISTA suite of technologies builds upon over a decade of research into robust, transparently distributed object-oriented operating systems; however, it is constructed from a completely new code base and design. While, because of its youth, it lacks the extensive production use of its predecessor technologies and corresponding breadth of affiliated applications, it does enjoy several significant advances in design and functionality. Although not an exhaustive list, the following illustrates some of the reasons why programmers benefit from using FARGOS/VISTA to develop new applications:

- Productivity: a 6 to 10-fold improvement in productivity using OIL2.
- Distributed across heterogeneous platforms: source code written for one platforms works on all the others. Architecture-neutral object code can even be generated and distributed without concern to the architecture of the target host.
- Robustness: there are no predefined limits from the very basic primitives all the way to complete applications, so no surprises due to unexpected limits being reached; memory management is performed by the underlying system
- Reusability: the object model encourages the implementation of generic components that can be reused by and extended.
- Reduced complexity: active objects break what would normally be a large, monolithic program into cooperating collections of well-focused objects. This also provides a natural mechanism for distributing functionality across multiple physical machines.

Model of Operation

As its name implies, the FARGOS/VISTA Object Management Environment implements an infrastructure that hosts and manages objects. The FARGOS/VISTA object model is intended to be being simple to understand, consistent and yet powerful. Many aspects of the object model will be familiar to programmers who have previously been exposed to object-oriented programming:

- A class defines both the variables that represent the state of a particular object (these are called instance variables) and the operations that can be performed against objects of a particular class (these operations are called methods).
- A class is uniquely named by three elements: a name space, the class name and a version Id. The FARGOS/VISTA Object Management Environment supports the simultaneous use of multiple versions of a class.

- A class can inherit from one or more classes (i.e., multiple inheritance is supported). The classes from which it inherits are called its *base* classes. From the perspective of its individual base classes, it is considered a *derived* class. The terminology of *super*- and *sub*-class is also commonly used; however, this document will use the terms *base* and *derived* as an aid to clarity since the similarity of the prefixes super and sub can create confusion when read quickly.
- A class must inherit from the base class **Object**. This can either be explicitly stated or implied as a property of inheriting from another base class that in turn eventually inherits from the class **Object**.
- Every class must have both a **create** and a **delete** method. While most classes have more methods, it is entirely possible to have a useful class that only implements these two methods.
- An object is said to be an instance of a class. It is a distinct collection of variables as defined by the class definition.

Every object is identified by a globally unique identifier, which is automatically generated at the time the object is created. This unique identifier is referred to as an object Id. Globally unique means across all machines, not just the address space into which the object is born.

The rules above pertain to the static nature of class definitions. The FARGOS/VISTA object model also includes operational aspects that are unconventional:

- Two objects can only interact through the sending of a message. This restriction is made visible in OIL2: it does not permit the use of pointers and thus the direct manipulation of another object's instance variables. In OIL2, a message is sent using the **send** statement. C++ programs can use the **OMEthread::sendMessage()** function or equivalent (e.g., **OMEapi::invokeMethod()**).
- In general, when a message is received for an object, the indicated method is executed. This process is called a *method invocation*. The indicated method may have a null body, which means that no code is to be executed. The **delete** method of many classes has this characteristic.
- If an object's method body is not null, then its execution is performed by a separate thread. This means that the runtime environment of OIL2 objects is one in which parallelism is supported at a fine level of granularity, namely that of a method invocation. If compared to conventional environments, this would correspond to a separate thread of execution being spawned for each function call. In contrast to convention programming models, this means that FARGOS/VISTA-based applications are composed of collections of active objects.
- By default, only one method can be active on an object at a time. This restriction enforces safe behavior by default and prevents race conditions, a common issue in multi-threaded environments. Except in very complex cases, programmers need take no action to disable the default behavior, but this capability is available (via the **allow()** and **alwaysAllow()** functions).
- If more than one method is active against an object, the other method cannot proceed until the currently active method is suspended. Again, the default is to enforce safe behavior and prevent race conditions, but this can be overridden by setting a thread as preemptable.

The rules presented above are elaborated upon later in this manual.

Development Languages

The programming language of choice for FARGOS/VISTA developers is Object Implementation Language 2 (OIL2) and most of the examples in this manual are presented using OIL2. There are several reasons for the preference of OIL2:

- The OIL2 language uses the FARGOS/VISTA Object Management Environment as its runtime environment, providing a one-to-one mapping for many features.
- It generally takes 6-10 times less code to express a routine in OIL2 than in C++.
- OIL2 source can be compiled to native object code or an architecture-neutral format that can be processed by a FARGOS/VISTA-based platform.

It is important to realize that programmers are not restricted to only use OIL2. For example, the FARGOS/VISTA Object Management Environment can also host applications completely written in C++. It can also be integrated with external applications by linking the FARGOS/VISTA Object Management Environment core as a library, thus adding FARGOS/VISTA capabilities to an existing application. Completely distinct applications residing in a separate address space or host can be integrated using the **OMEapi()** library.

Deploying New Applications

On a given host, the FARGOS/VISTA Object Management Environment is normally established by running a single process and all FARGOS/VISTA-based applications reside within this single process. This is one aspect of the power of the FARGOS/VISTA-environment: distinct applications written by various programmers live within the same address space and can operate in isolation or cooperate as needed. It is also possible to use a separate process for each mission-critical application; however, the overhead of context switches between separate address spaces is then incurred.

New FARGOS/VISTA application code is deployed in one of three primary ways:

- The application's native object code is linked with the FARGOS/VISTA Object Management Environment library and a custom executable is created.
- The application's native object code is converted to a shared object file and dynamically loaded into a running Object Management Environment process. This is only possible on platforms whose native operating systems support dynamically linked executables. Most operating systems that support this capability require that the dynamically loaded object code by stored in a file.
- The application is compiled into an architecture-neutral object code (a capability of the OIL2 compiler) dynamically loaded into a running Object Management Environment. The object code can be retrieved from a file on the local system or a string in memory.

The use of architecture-neutral object code has a few advantages. During the development phase of an application, the time needed for a programmer to compile, load and start execution of a new version of the code under development can take less than a second. With such a high level of interactive performance, the time required to compile and link new executables disappears as an issue. Architecture-neutral object code also permits a developer to distribute a single object file without having to be concerned about the end user's target platform.

Native object code is the format of choice when the goal is to achieve maximum performance. It can also be used to enforce price differentials for different platforms (for example, Linux-based workstations vs. OS/390-based mainframes). Native object code is also a requirement in some situations, such as providing an integration layer to an existing library.

Note: the quality of native code can be improved if the target environment's characteristics are well known and uniform. For example, most code for current Intel processors is actually generated so that it will run on an Intel 386 CPU and upwards. If one knows that the target is actually an Intel Pentium III, then performance gains can be realized by taking advantage of CPU-specific instructions. The significant drawback is that the resulting object code will not run on earlier Intel processors or those manufactured by AMD. The same issues exist in each long-running CPU architecture (e.g., IBM System/360/370/390, Sun SPARC/UltraSPARC, IBM Power/Power2/PowerPC). Consequently, most applications are compiled to use the set of CPU instructions that represent the lowest common denominator architecture[1].

[1] FARGOS Development, LLC can provide specialized builds upon request.

2. The VISTA Daemon

The standard FARGOS/VISTA Object Management Environment is made available as the executable program **vista**. Note that on some native operating systems, the executable is required to be named **vista.exe,** but it invariably can still be invoked using the common name **vista**. The executable can be run directly from the command line or started as a long-running daemon in the background. On Microsoft Windows NT-derived systems, the **OMEregNTserv.exe** utility can be used to start the **vista.exe** process as a managed service.

The Boot Process

The **vista** executable is normally invoked with a single argument that specifies the name of a file that will be processed by the class **CreateObjects**. The boot process of a VISTA daemon takes place in two basic steps: the system is initialized and then it runs threads until no more work can be done. The initialization is performed by the **OMEinitSystem()** function and performs the following sequence of actions:

1. All statically linked modules are initialized.
2. All mandatory native shared object modules are dynamically loaded and initialized.
3. An object of class **ObjectCreator** is created and registered as the local service *ObjectCreator*.
4. An object of class **ShutdownService** is created and registered as the local service *ShutdownService*.
5. An object of class **CreateObjects** is created and provided with the file name that was passed as an argument to the **vista** executable (or the default *vista.rc* is none was specified).

Once the system is initialized, execution of threads can begin. This is normally performed by calling the function **OMEmainLoop()**, which returns only when no more threads are active and no input/output or timer events remain.

The FARGOS/VISTA Software Development Kit contains a default **VISTAOMEmain()** function that is very similar in appearance to the following example:

```
#include <OMEruntime.h>

int VISTAOMEmain(int argc, char *argv[], char *envp[])
{
        char            *rcFileName;
        int             i;

        rcFileName = "vista.vrc";        // set default...
        for(i=1;i<argc;i++) {
                if ((strcmp(argv[i], "-d") == 0) ||
                         (strcmp(argv[i], "+d") == 0)) {
                        ++i;    // skip debug flag argument...
                        continue;
                }
                if ((*argv[i] != '-') && (*argv[i] != '+')) {
                        rcFileName = argv[i];
                }
        }

        OMEinitSystem(rcFileName, argc, argv, envp);
        OMEmainLoop();
        return (0);
}
```

The initial command file is critical to the operation of the **vista** executable since it details the starting configuration of the environment. On platforms that support it, the script can itself be made executable and then invoked as a command in its own right. Consider the following example:

```
#!/usr/local/bin/vista
LoadObjectFile clMyApp.so
MyApplication arg1
```

On most modern Unix systems, if the text above is placed into a file *myapp*, marked as executable and then invoked, the **vista** daemon at */usr/local/bin/vista* will be started and it in turn will process the object creation directives specified. A similar effect can be achieved with desktop-based operating systems by creating a file with a particular file suffix (e.g., "*.vrc*" for **vista** *rc*) that is associated with the **vista** executable for that platform. Such an association is stored in the registry by the automated installation programs released by FARGOS Development, LLC for variants of Microsoft Windows (95/98/NT 4/2000/XP).

Linking a Custom VISTA Executable

In contrast to conventional application development, which yields a standalone executable, most FARGOS/VISTA-based applications expect to be co-resident with other applications inside the address space of a single FARGOS/VISTA Object Management Environment. To enable this, developers normally distribute a command file to be processed by **CreateObjects** and an appropriate dynamically loadable module. The dynamically loadable module is often an OIL2 Architecture-Neutral Format (OIL2 ANF) file, but it can also be a shared object file readable by the host's native operating system. There are times, however, where a developer may wish to ship a self-contained executable that has been customized for the application in question. The FARGOS/VISTA Software Development Kit for the target platform is a prerequisite for creation of a custom executable that includes statically linked modules or will automatically load dynamically loadable object files.

There are thus two distinct modes in which native object code can be generated and four modes in which it can be used:

Table 1

Compile Mode	Link Mode
Conventional object (.o, .obj)	Statically linked to form executable.
Shared object (.so, .dll)	Automatically dynamically loaded by the executable upon start of execution
Shared object (.so, .dll) or OIL2 Architecture Neutral Format (.o2o)	Explicitly loaded by **LoadObjectFile**, typically during the processing of a script by **CreateObjects**.
	Automatically loaded on demand via facilities provided by the **AutomaticClassLoader** service.

To create a custom **vista** executable, the following steps should be followed:

1. If the application is written in OIL2, compile the application to C++ source.

2. Compile the application's C++ source code using the native C++ compiler, yielding an object file for the target platform.
3. Generate a module list file for new executable by using the **mkModuleList** command. It takes as arguments the names of all the object files (both conventional and shared) needed to create the executable. In this scenario, **mkModuleList** would be passed the object files of the application and the FARGOS/VISTA libraries. The **mkModuleList** program outputs a C++ source file; typically, this is saved as *moduleList.cpp*.
4. Compile the output from **mkModuleList** (typically, *moduleList.cpp*) using the native C++ compiler.
5. Link all of the conventional objects (the application object files, the FARGOS/VISTA libraries and the *moduleList* object) together to create a new custom executable.

The **mkModuleList** program recognizes several different file types and generates an appropriate C++ source file that defines the module initialization table. If the file is a shared object file (.so) or dynamically loaded library (.dll), the module initialization table is created such that the file is automatically dynamically loaded; otherwise, the object file is assumed to be statically linked into the executable and the file's initialization routine is called without first dynamically loading the object code.

For a given object file, **mkModuleList** looks for symbols that begin with the pattern "*INIT_DECLARE_*". The convention used is that the "*INIT_DECLARE_*" text is suffixed with the name of the file minus its file type suffix. For example, a file *clMyApp.oil* would be compiled to *clMyApp.cpp* by the OIL2 compiler. The native C++ compiler would compile *clMyApp.cpp* to an object file named *clMyApp.o* or *clMyApp.obj* (depending on the host platform). The name of the file minus its file type suffix is "*clMyApp*", so the full name of the symbol would be "*INIT_DECLARE_clMyApp*". The OIL2 compiler generates such initialization routines automatically; developers creating handcrafted code should follow the convention specified above.

An example initialization routine for a module appears below:

```
#ifdef _MSC_VER /* Microsoft Visual C++ */
extern "C" __declspec(dllexport) void INIT_DECLARE_clHTTP()
#else
extern "C" void INIT_DECLARE_clHTTP()
#endif
{
        OMEclass        *classRecord;
        int             i;

        INIT_CONSTANTS();

        // Define classes
        classRecord = OMEdefineNewClass("Standard",
                "HTTPdaemon",
                0, sizeof(class OIL2_CL_Standard_HTTPdaemon_0),
                0, 0,
                OIL2_CL_Standard_HTTPdaemon_0::ALLOCATE_AND_INIT,
                OIL2_CL_Standard_HTTPdaemon_0::FREE_STORAGE);
        classRecord->setStorageDescription(OIL2_CL_Standard_HTTPdaemon_0_varTabl
e);
        classRecord->inheritFromClass("", "Object", 0, 0);
        classRecord->resolveLinkages();
}
```

The following *Makefile* provides an illustration of the generation of a custom **vista** executable using the FARGOS/VISTA Software Development Kit. It assumes the use of GNU make, which has been ported to all of the platforms supported by FARGOS/VISTA. The conditional directive **ifeq()** is used to handle some platformspecific configuration; developers can use the native **make** program associated with a particular platform by removing the non-essential portions.

```
# (C) Copyright FARGOS Development, LLC 1999.  All rights reserved.

# *** NOTE *** External variables used
# OBJ_SUFFIX - typically .o or .obj depending on platform
# LIB_SUFFIX - typically .a or .lib depending on platform
# EXE_SUFFIX - typically not defined (or null) or .exe depending on platform
# DYNAMICALLY_LOAD_CLASSES - defined if most core classes are to be dynamically
#  loaded.  Runtime always has capability to load more dynamically
#  regardless of the definition of this variable

DYNAMICALLY_LOAD_CLASSES=0
USE_DYNAMIC_LIBRARY=0

.PRECIOUS: .cpp
.SUFFIXES: .so .dll .obj

# C++ compilation rules
%.${OBJ_SUFFIX} : %.cpp ; ${CC_PLUSCOMP} ${OPTIMIZE} -D_REENTRANT –
I${VISTA_ROOT}/include ${CPLUSFLAGS} -c $<
# end C++ rules

# OIL2 -> C++
ifeq (${OBJ_SUFFIX},obj)
%.cpp : %.oil ; ${VISTA_ROOT}/${VISTA_UNAME}/bin/oil2.BAT $<
else
%.cpp : %.oil ; ${VISTA_ROOT}/${VISTA_UNAME}/bin/oil2 $<
endif
# UNIX .o to shared object
#
# magic for making the .so for ldLoading... sample:
# g++ -shared -o OMEfileDescriptor.so OMEfileDescriptor.o
#
.o.so:
   g++ -shared -o $@ $<

# Windows .obj to DLL
.obj.dll:
   link /nologo /force /dll /out:$@ $<


ifeq (${VISTA_UNAME},OpenBSD)
   export DONT_USE_PTHREADS=1
   OPTIMIZE += -DDONT_USE_PTHREADS
endif


ifdef DONT_USE_PTHREADS
   PTHREAD_LIB=
else
   PTHREAD_LIB=-lpthread
   ifeq (${VISTA_UNAME},SunOS)
         PTHREAD_LIB += -lrt
   endif
endif
DYNAMIC_LOAD_LIB=
ifeq (${VISTA_UNAME},Linux)
   DYNAMIC_LOAD_LIB=-ldl
endif
ifeq (${VISTA_UNAME},SunOS)
   DYNAMIC_LOAD_LIB=-ldl -lsocket -lnsl
endif

OME_CORE_STATIC_LIB=OMEcore.$(LIB_SUFFIX)
OME_CORE_DLL=OMEcore.$(DLL_SUFFIX)


# ** NOTE ** ORDER IS IMPORTANT--THIS SECTION MUST APPEAR BEFORE DEFINITION OF
# DLOBJS
ifeq (${DYNAMICALLY_LOAD_CLASSES},1)
O_SUFFIX=${DLL_SUFFIX}
else
```

```
O_SUFFIX=${OBJ_SUFFIX}
endif

# OIL-implemented class objects that must be present in every binary
# prior to the dynamic loading of any others.
# This should be a relatively small set and need not be adjusted
# if a statically linked executable is produced
BASIC_CLASS_OBJS=clMyStatic1.${OBJ_SUFFIX}

# OIL-implemented class objects that can be either statically linked or
# dynamically loaded.  The bulk of the classes appears here.
# NOTE:  THESE SHOULD ALL USE O_SUFFIX, NOT OBJ_SUFFIX!
DLOBJS=clMyDL2.$(O_SUFFIX)

OBJS=$(CORE_OBJS) $(BASIC_CLASS_OBJS)
ifeq (${DYNAMICALLY_LOAD_CLASSES},1)
   ALL_OBJS=$(OBJS)
   AUX_OBJS=$(DLOBJS)
else
   ALL_OBJS=$(OBJS) $(DLOBJS)
   AUX_OBJS=
endif
ifeq (${USE_DYNAMIC_LIBRARY},1)
         OME_CORE_LIB=$(OME_CORE_DLL)
else
         OME_CORE_LIB=$(OME_CORE_STATIC_LIB)
endif


VISTA_OBJECTS=vista.${OBJ_SUFFIX} moduleList.${OBJ_SUFFIX} $(OME_CORE_LIB) $(LIBS)

all:     vista${EXE_SUFFIX}

# UNIX binary...
vista: $(VISTA_OBJECTS)
   ${CC_PLUSCOMP} ${OPTIMIZE} -rdynamic -o $@ $(VISTA_OBJECTS) $(DYNAMIC_LOAD_LIB)
$(PTHREAD_LIB)

# Windows binary
vista.exe: $(VISTA_OBJECTS)
   cl /Fe$@ $(OPTIMIZE) $(VISTA_OBJECTS) /link ws2_32.lib advapi32.lib odbc32.lib

ifeq (${VISTA_UNAME},Windows)
moduleList.cpp:      $(ALL_OBJS) $(AUX_OBJS)
   mkModuleList.bat $(ALL_OBJS) $(AUX_OBJS) > $@
else
moduleList.cpp:      $(ALL_OBJS) $(AUX_OBJS)
   sh mkModuleList catSym $(ALL_OBJS) $(AUX_OBJS) > $@
endif
```

3. Defining New Classes

Object code generated by an OIL2 compiler always includes all of the mechanisms needed to automatically register the classes defined within the object code module. Consequently, programmers that avail themselves of the opportunity to write their applications in OIL2 need not be concerned with the procedure of registering classes.

On the other hand, programmers that write handcrafted classes must register such classes and their associated methods with the FARGOS/VISTA Object Management Environment before they can be used. Class definitions are normally registered using the C++ function **OMEdefineNewClass()**. Once defined, the meta data for the class should be set using the C++ function **OMEclass::setStorageDescription()**. The list of inherited classes must be specified using appropriate calls to **OMEclass::inheritFromClass()**. When completely defined, **OMEclass::resolveLinkages()** must be called to attempt to link the class against any inherited classes. The operation may fail because a needed class is not yet available, but if the missing class is subsequently loaded, the linkage will automatically be completed the first time an object of this class is created.

```
classRecord = OMEdefineNewClass("Experimental", "GoogleSearchService", 0,
   sizeof(class OIL2_CL_Experimental_GoogleSearchService_0), 0, 0,
   OIL2_CL_Experimental_GoogleSearchService_0::ALLOCATE_AND_INIT,
   OIL2_CL_Experimental_GoogleSearchService_0::FREE_STORAGE);
classRecord->setStorageDescription(
   OIL2_CL_Experimental_GoogleSearchService_0_varTable);
classRecord->inheritFromClass("", "XMLsupport", 0, 0);
classRecord->inheritFromClass("", "SOAPservice", 0, 0);
classRecord->resolveLinkages();
```

After a class is defined, its associated methods must next be declared. This is done using the C++ function **OMEdefineNewMethod()**. As noted above, all of this work is automatically performed by object code files that are generated by an OIL2 compiler.

```
for(i=0;i<sizeof(methodTable) / sizeof(OMEmethodDefinition);i++) {
   OMEdefineNewMethod(methodTable[i]);
}
```

4. Security

There are three primary reasons why security is a paramount concern of the FARGOS/VISTA environment:

- FARGOS/VISTA implements a transparently distributed environment. Because not all hosts that participate in a distributed system are under control of the same administrative authority, resources must be protected from unauthorized use from remote hosts.
- Many FARGOS/VISTA-based applications are long-running services that perform activities on the behalf of many different users. Not every user is a privileged user and users should be prevented from obtaining or modifying information that does not belong to them.
- Usually, a given FARGOS/VISTA Object Management Environment will host several applications simultaneously. Such applications need to be protected from unauthorized use.

Access Control Lists

All objects and threads within a FARGOS/VISTA Object Management Environment are protected by access control lists. Access control lists can specify permissions at a very fine level of granularity, namely per-user and per-method. Every access control list has a default permission setting that is used if there is no entry for the active user. The OIL2-callable function **makeDefaultACL()** creates an access control list that permits the current user complete access to an object and denies access to all others. It in turn uses the C++ routines **OMEcreateACL()** and **OMEaddToACL()**.

Access control lists are associated with an object Id. Recall that one of the fundamental rules of the FARGOS/VISTA object model is that every object has a globally unique Id (discussed above in the section entitled "Model of Operation"). Without violating this requirement, the FARGOS/VISTA Object Management Environment permits multiple, globally unique object Ids to refer to the same physical object. Each such object Id will be associated with a different access control list. By handing out an object Id with an appropriately constructed access control list, an application can maintain extremely fine-grained access control to its methods. Taken to the extreme, a given object Id can permit a single user to invoke only one of the methods associated with an object. An access violation will occur if the object Id is used to invoke a different method or used by a different user.

Technically, there is no notion of the "owner" of an object. Access privileges are completely controlled by having possession of an object Id that permits a user to perform desired actions.

Access control lists are directly used by programmers in two places. The first is when an object is created by sending a **createObject** (or **createObjectAndNotifyWhenDone**) message to the **ObjectCreator** object. The second is when a new object Id is created using the OIL2-callable function **createNewOIDthatOnlyAllowsOthers()** (or in C++ when making a new **OMEoid**). An example usage of both cases appears below:

```
acl = makeDefaultACL();
readObj = send "createObject"("ReadAndProcessFile", acl,
      thisObject, ioObj, acl, MAX_LINES_PER_SEND)
      to ObjectCreator;

m[0] = "releaseThread";
sleepingThread = createNewOIDthatOnlyAllowsOthers(thisThread, m);
allow("processLine");
send "suspendThread" to thisThread;
sleepingThread = nil;
```

Users

While threads are manipulated in the same fashion as objects, threads are not objects and the class **Thread** does not inherit from the class **Object**. Whereas objects do not have "owners", threads are always associated with some user. The user may be known to the local host operating system or a logical user known only within the FARGOS/VISTA Object Management Environment. A thread can change its associated user using the OIL2-callable function **becomeUser()** or the C++ function **OMEthread::setUserId()**. It is common to implement a service that will perform actions on behalf of anonymous users—a current well-known example would be an HTTP (world wide web) server. For strengthened security, the OIL2-callable function **becomePseudoUser()** can be used to create a unique, albeit temporary, logical user identity and associate it with the threads that perform the anonymous user's request.

Encryption

When two FARGOS/VISTA processes (i.e., distinct address spaces) communicate, all traffic is encrypted. Each side of a communications link generates a random 128-bit session key. This means that the session key used to send data is distinct from the session key used to decrypt incoming data. Whenever random data is required, FARGOS/VISTA makes use of random number generator devices if they are supported by the underlying host's operating system. The randomly generated session keys are communicated using public key encryption and a multi-step protocol, so they never appear in the clear and man-in-the-middle attacks are detected. If a man-in-the-middle attack is attempted, only information related to the randomly generated session key is exposed and, because the attack is detected, the connection will never be completed and no data will be exposed to the attacker. The default symmetric encryption algorithm is *Rijndael*, which was ultimately selected as the Advanced Encryption Standard at the end of 2000 by the United States National Institute of Standards and Technology and documented as FIPS-197.[2]

A collection of C++ cryptographic functions is defined in the *OMEcrypto.h* header file. These deal with public key exchange of session keys, generation of random byte strings, Secure Hash Algorithm 1, and the encryption and decryption of blocks of data. OIL2 programs interface to these facilities by using **makeRandomKey()**, **SHA1hash()**, **initializeCipher()**, **encryptMessage()** and other associated functions.

5. Input / Output Transport Schemes

Many conventional applications that work with byte streams or packets need to take into account the underlying characteristics of the input/output mechanism. For example, files are opened differently than TCP sockets. The FARGOS/VISTA infrastructure provides a common, high-level and extensible mechanism that permits generic input/output routines to work with a wide variety of transport mechanisms. Users and applications identify files and sockets using a form of Universal Resource Locator (URL).

Developers can register new or custom transport schemes by using the **OMEregisterIOscheme()** function. Any FARGOS/VISTA application that links with or dynamically loads such an extension can make use of the new transport schemes without requiring recompilation of existing code. The standard FARGOS/VISTA infrastructure provides support for the schemes that are listed in Table 2. Note that one should not expect Unix file domain schemes to be supported on non-Unix platforms. A given host may not have all supported protocol stacks configured into its kernel—thus while an IP Version 6 transport scheme may be understood by the FARGOS/VISTA Object Management Environment, use of IP version 6 may be impossible on a given host due to restrictions imposed by its kernel configuration.

It is reasonable to assume that a given host will successfully support the “file:”, “tcp4:” and “udp4:” transport schemes; availability of the remainder is dependent on the kernel configuration of individual hosts.

[2] Earlier releases of FARGOS/VISTA used *twofish*, which was one of the Advanced Encryption Standard finalists.

Table 2

Scheme Prefix	Transport Description
File:	access files in the local hosts file system
unix: unixstream:	Unix file-domain stream connections
unixdatagram:	Unix file-domain datagram connections
tcp: tcp4:	listen for or establish IP version 4 TCP-based connections
tcp6:	Like tcp4:, but for IP version 6
udp: udp4:	IP version 4 UDP ports
udp6:	Like udp4:, but for IP version 6
raw: raw4:	IP version 4 raw network socket
raw6:	Like raw4:, but for IP version 6
ipx:	Novell IPX (datagrams)
spx:	Novell SPX (sequenced packets/streams)

All open file and socket descriptors are maintained by an object that is an instance of an appropriate class derived from the C++ class **OMEioDescriptor**, which is defined in the C++ header file *OMEioObjects.h*. Most FARGOS/VISTA Object Management Environment applications make use of the class **IOobject** to access these facilities; external C++ applications can use the **OMEopenURL()** function to create an appropriate object of class **OMEioDescriptor** (or a derived class).

6. Data Encoding

FARGOS/VISTA runs on many different platforms and supports communication between all of them. This requires that data can be exchanged between two distinct platforms. The FARGOS/VISTA infrastructure provides a common, high-level and extensible mechanism for encoding and decoding all FARGOS/VISTA data types. Each encoding scheme is identified by a 32-bit version Id. The default encoding schemes present in all FARGOS/VISTA components are version 1 and its compressed counterpart, version 2.

Support for new encoding schemes is added using the C++ function **OMEdefineEncodeRoutinesForVersion()**, which is defined in the C++ header file *OMEencode.h*. The registration of the default version 1 and version 2 encoding routines is conveniently performed by the C++ function **OMEloadVersion1Encodings()**, which is called by **OMEinitSystem()**.

The encoding of complex data structures and/or multiple items of data is handled in an optimal fashion by the C++ class **OMEencodeBuffer()**. A pointer to an **OMEencodeBuffer()** is taken as the argument to all **OMEtype:: encode()** routines. When all the elements of data have been encoded, it can be serialized into one large string using the **OMEncodeBuffer::condenseIntoString()** function. This avoids the repeated concatenation that is common with approaches that are more conventional. An example of the ease of use is provided below:

```
    OMEtype     result;
    OMEencodeBuffer *encodeBfr;
    int             rc;
    OMEstring       *encodedData;

    encodeBfr = new OMEencodeBuffer(1);
    rc = data.encode(encodeBfr);
          if (rc != 0) { // failed…
          return (result);
    }

    encodedData = encodeBfr->condenseIntoString(1);
    result = encodedData;   // take ownership
    delete encodeBfr;
    return (result);
```

Decoding of previously encoded data is performed using the C++ static function **OMEtype::decode()**.

Applications written in OIL2 make use of the **encodeData()** and **decodeData()** functions.

String Encoding Formats

Whereas the **encodeData()** and **decodeData()** functions deal with any kind of FARGOS/VISTA data type, string data represents a special case for which there are several additional available encoding formats. The table below lists several:

Encode		Decode		Notes
C++	OIL2	C++	OIL2	Binding
OMEcompressString()	compressString()	OMEuncompressString()	uncompressString()	
OMEgzipString()	gzipString()	OMEgunzipString()	gunzipString()	RFC 1592
OMEbinaryToBase64()	asciiToBase64()	OMEbase64ToBinary()	base64ToASCII()	RFC 1521
OMEbinaryToHex()	makeAsHexString()	OMEhexToBinary()	hexToBinary()	

7. External Applications

Conventional applications can be externally integrated with the FARGOS/VISTA Object Management Environment using the C++ class **OMEapi**, which is defined in the header file *OMEapi.h*. The same mechanism and protocol is used to interface external applications as is used by inter-FARGOS/VISTA Object Management Environment connections. This means that the same security mechanisms are enforced and the endpoint is represented by a **PeerConnection** object. When a connection is established between the external application and a FARGOS/VISTA Object Management Environment, an object Id mapping is created within the Object Management Environment that corresponds to the external application. Subject to access control, any message sent to that object Id from any interconnected peer is automatically encoded and forwarded to the external application. Such messages can be retrieved using the **OMEapi::importInvocation()** function. Responses or unsolicited requests can be sent from the external application to any object within the connected peer systems using the **OMEapi::invokeMethod()** function.

The standard FARGOS/VISTA Object Management Environment class **RegisterTemporaryService** provides a convenient mechanism for automatically deregistering a service associated with an external process. The **createTemporaryObject** method of class **PeerConnection** can be used for automatic cleanup by externally attached applications with more sophisticated needs.

The code fragment below illustrates the usage of the **OMEapi**. Source code to complete examples can be found at http://www.fargos.net/examples.

```
OMEinitDebugFlag();
OMEloadVersion1Encodings();
OMEregisterStandardSocketSchemes();

api = new OMEapi(acl, addr);
count = 0;
while (1) {
   api = new OMEapi(acl, addr);
   rc = api->establishConnection(connectionUserInfo);
   if (rc == 0) break;     // success..
   delete api;
   api = 0;
   if (count == 3) {
         std::cerr << "OMEpersistd:  could not connect to " << ad
dr << "\n";
         return (2);
   }
   ++count;
#ifdef _WIN32
   SleepEx(3 * 1000, FALSE);
#else
   sleep(3);
#endif
}
// perform service...
do {
   rc = api->importInvocation(methodName, args, &fromObj,
   &targetObj, &context, &user);
   if (rc != 0) break;
   // validity check:  could verify if targetObj == thisObject
   if (OMEdebugFlag & OMEdebugLogLevel1) {
         std::cerr << "methodName=" << methodName << "\n";
         std::cerr.flush();
   }
   if (*methodName.value.s == "saveObject") {
         rc = doSaveObject(args, fromObj, context, user);
         args.initializeAsType(OME_ARRAY);
         args[(uint32) 0] = rc;
         rc = api->invokeMethod(fromObj, replyMethodName, args);
   } else {
         std::cerr << "unrecognized methodName=" << methodName << "\n";
         std::cerr.flush();
   }
   if (rc != 0) break;
} while (OMEstopFlag.value.ui == 0);
delete api;
```

8. Working with Objects

All objects maintained by a FARGOS/VISTA Object Management Environment must have the class **Object** as their ultimate base class. Every object is identified by a globally unique object Id, which serves as a handle. Objects are created by sending a **createObject** message or equivalent to the **ObjectCreator** object. Space for the instance variables of each class are allocated using a storage allocation function, which is potentially class-specific but frequently realized using a generic function associated with the implementation language. Objects are initialized by invoking in order the **create** methods of the base classes and derived classes. The process always begins with the **create** method of class **Object** and always ends with the invocation of the **create** method of the derived class that was specified as an argument in the **createObject** request. That **create** method is provided the arguments that were passed in the **createObject** request. An object is deleted by sending it a **deleteYourself** message, which is implemented by the base class **Object**. When a **deleteYourself** message is processed, a series of **delete** method invocations are queued such that the object is torn down in the exact inverse order of that in which it was built up. Thus, the **delete** method of class **Object** is always the last invoked. Storage is ultimately recovered using a storage reclamation function, which can be class-specific.

9. Working with Threads

One of the unconventional characteristics of the FARGOS/VISTA Object Management Environment is that a new thread is created for every method invocation. While threads are not objects, each thread is identified by an object Id and it can be sent messages in the same fashion as an object. OIL2 programs always view a thread as being an instance of class **Thread**. On the other hand, C++ programs can also work with the underlying implementation in the C++ class **OMEthread**. Since threads can be sent messages just like objects, it is entirely possible for an object on a remote peer system to send a message that wakes up or terminates a local thread.

Each FARGOS/VISTA Object Management Environment usually supports 3 distinct threading technologies: two are proprietary ultra-high performance techniques that work without any support from the host's kernel. The third is the native kernel's support for lightweight processes. If such support is available from the host operating system, the FARGOS/VISTA Object Management Environment can automatically exploit symmetric multiprocessor hardware. Since every method invocation is a logical separate thread of execution, the opportunities for parallelism in a FARGOS/VISTA-based application are frequent, despite requiring no action to be taken by the programmer.

Although thousands of threads may be active simultaneously within a given address space, by default, only one method can be active upon a given object. If a message is sent to an object while a thread is already active upon it, the method invocation will be queued. If additional messages are received, they too will be queued and the order in which they arrived will be maintained. Normally, the first queued thread will be started when the currently active thread terminates. This behavior can be overridden by allowing an invocation of a particular method to proceed. The **allow()** function permits one such method invocation to be started and the **alwaysAllow()** function permanently permits all such method invocations to be started. C++ programmers are encouraged to use the OIL2 interfaces, but they can use an underlying **OMEobjectInstance::allowMethod()** function as illustrated below:

```
    OMEoid &thisObj = thread->getThisObject();
    const OMEobject *obj = thisObj.getInstanceInMemory();
    ((OMEobjectInstance *) obj)->allowMethod(*methodName.value.s);
```

There is another level of safety provided by default: although multiple threads may be allowed to be active on an object, normally only one thread associated with the object can be running at a time. Simultaneous execution of threads on an object is possible if a thread is marked as preemptable. OIL2 applications achieve this by sending the thread a **setAsPreemptable** message:

```
   send "setAsPreemptable" to thisThread;
```

C++ programmers can perform the same method invocation or just use the function **OMEthread::setAsPreemptable()**:

```
   thread->setAsPreemptable();
```

Threads can be terminated, put to sleep or woken up by sending them messages. For example:

```
   send "terminateThread" to thisThread;     // logical equivalent to exit
statement
   send "suspendThread" to thisThread;       // put to sleep
   send "releaseThread" to sleepingThread;// wakeup
```

C++ programmers can use the equivalent member functions of the C++ class **OMEthread**. C++ programmers should always use the **OMEthread::exitRoutine()** function when exiting from a method body as this routine also automatically handles the case where a method was called instead of being invoked.

10. Reflection and Meta Data

All data manipulated by OIL2 programs is tagged and its type can be determined by the **typeOf()** function. In a similar fashion, information about an object's class can be retrieved by sending it appropriate messages. These methods are defined in the class **Object**, which is the base class for all classes (except **Thread)** in a FARGOS/VISTA Object Management Environment. The **isOfClass** method may be the most commonly used of such inspection methods.

The FARGOS/VISTA Object Management Environment also supports a powerful facility called reflection[3], which allows the behavior of an object to be overridden by another object. In formal computer science theory, it permits the expression of behavior using the higher-level facilities of the environment. In practice, it can be use to dynamically implement bug fixes, debug and trace method invocations against an object, modify and extend the behavior of applications for which one only has the object code, etc. Reflection on a per-object basis is obtained by sending an object a **setMeta** message.

Reflection on a class-specific basis is also possible by sending a **setClassMetaObject** message to the **ObjectCreator** object. . This technique can be used for on-demand conversion of objects:

1. The conversion application sets itself as the meta object of the old version of the class.
2. Every time an object that is an instance of the old version of the class is referenced, the invocation is passed to the class-specific meta object, which in this case would be the conversion application.
3. The conversion application retrieves the old data and uses it to create an object of the new version of the class.
4. The old object is deleted.
5. References to the new object proceed unimpeded because there is no meta object associated with the new version of the class.

[3] J. Ferber, "Computational reflection in class based object-oriented languages", *OOPSLA 1989 conference proceedings*, pp. 317-326, 1989

11. Getting Started

The easiest way to gain familiarity with the FARGOS/VISTA Object Management Environment is to create a simple class, load it and use it. Consider the following OIL2 source program:

```
%include <OMEcore.o2h>

class Local . FirstDemo {
   any   args;
} inherits from Object;

FirstDemo:create(int delete)
{
   args = argv;      // save copy
   display("In FirstDemo:create, argc=", argc, "\n");
   if (delete == 1) {
         send "deleteYourself" to thisObject;
   } else {
         display("Staying around\n");
   }
}

FirstDemo:delete()
{
   display("FirstDemo:delete obj=", thisObject, "\n");
   display("Argument list was ", args, "\n");
}
```

Assume the above was placed into a file named *clFirstDemo.oil*. The following command would compile it into the architecture-neutral format (OIL2 ANF) and place the result in the file *clFirstDemo.o2o*:

```
% oil2_parse –oil2 clFirstDemo.oil
```

To test the application, the *clFirstDemo.o2o* file needs to be loaded into the FARGOS/VISTA Object Management Environment and an object of class **FirstDemo** needs to be created. The command file below does just that:

```
LoadOIL2File file:clFirstDemo.o2o
FirstDemo 1 arg2 arg3 arg4
```

Assuming the above was placed into a file called *demo.vrc*, the application could be executed by issuing the following command:

```
   vista demo.vrc
```

One would expect to see output similar to:

```
In FirstDemo:create, argc=4
FirstDemo:delete obj=[1:0:1697257140:(2467193760|4294302185|50444)]
Argument list was {
  [0] = int32 1
  [1] = string "arg2"
  [2] = string "arg3"
  [3] = string "arg4"
}
```

Applications that reside within a FARGOS/VISTA Object Management Environment are frequently developed and deployed in a fashion similar to the above. A collection of introductory examples can be found in the tutorial *An Introduction to Programming Using OIL2*. Some application may be statically linked to the runtime and automatic on-demand loading of classes can eliminate explicit preloading of class implementations. Applications may be initiated as a result of requests that originated on a remote peer system rather than from the initial *rc* file read at the start up of the local Object Management Environment.

12. Object Management Environment Classes

Most OIL2 programmers take advantage of the compiler's support for maintaining documentation with the source code. FARGOS/VISTA distributions normally include a collection of HTML files that describe class interfaces. By convention, these files are placed under the *$VISTA_ROOT/classDoc* subdirectory (see the *FARGOS/VISTA Installation Guide* for details). Links to many standard classes are provided below; programmers should always check their actual distribution for the most current information.

Examples of the use of many of these classes are demonstrated in the programming guide *FARGOS/VISTA Examples*.

13. OIL2 Class Documentation

Classes in Namespace Experimental

- ***PersistentDatabaseViaSQL***

Classes in Namespace Local

- ***ConnectAndForward***
- ***ConnectAndSlowlyForward***
- ***ForwardConnection***
- ***HTTPget***
- ***HTTPuserAdmin***
- ***POP3server***
- ***ReadCSVfile***
- ***SOAPservice***
- ***SlowlyForwardConnection***
- ***StarshipHTTPsetup***
- ***WSDLtoOIL2***
- ***XMLsupport***

Classes in Namespace Standard

- ***AcceptConnection***
- ***AcceptPeerConnections***
- ***AllocateSessionID***

- *AnnounceServices*
- *AutomaticClassLoader*
- *ClassLocator*
- *ConnectToPeer*
- *CreateObjects*
- *CreateReplicaHTTPsession*
- *CreateReplicasOnServers*
- *DNSconnection*
- *DNSresolver*
- *EstablishConnection*
- *ExecProcess*
- *ForwardReply*
- *HTTP_SSIprocessor*
- *HTTPcachedFile*
- *HTTPcachedObject*
- *HTTPcommonLogFormat*
- *HTTPcreateApplicationSession*
- *HTTPcreateObjectBrowserSession*
- *HTTPdaemon*
- *HTTPdisplayObject*
- *HTTPextendedLogFormat*
- *HTTPfastReceive*
- *HTTPobjectBrowser*
- *HTTPprotectedDirectory*
- *HTTPpurgeCache*
- *HTTPrawCachedFile*
- *HTTPredirect*
- *HTTPreplacedText*
- *HTTPreplicaClientProxy*

- *HTTPreplicaStateVariable*
- *HTTPrequest*
- *HostTable*
- *IOobject*
- *JobController*
- *LoadOIL2File*
- *LoadObjectFile*
- *MakeOIL2ANFprofile*
- *NameServerDirectory*
- *NegotiatePeerConnection*
- *Object*
- *ObjectCreator*
- *ParseParameterFile*
- *PeerConnection*
- *PeerRegistry*
- *PersistencePageIn*
- *PersistenceService*
- *PersistenceTemporaryService*
- *PersistentObject*
- *ReadAndProcessFile*
- *ReadBuffer*
- *ReadMIMEtypeFile*
- *RegisterPoolMembership*
- *RegisterReplicaHTTPclass*
- *RegisterTemporaryService*
- *ReplicaClientProxy*
- *ReplicaHTTPsession*
- *ReplicaServer*
- *SQLviaODBC*

- ***SendFile***
- ***SendMailViaSMTP***
- ***ShutdownService***
- ***Thread***
- ***TimerEvent***
- ***TraceInvocations***
- ***URLdirectory***
- ***URLfileLoader***
- ***URLprotectedFile***
- ***WebDAVcollection***
- ***WebDAVfacility***
- ***WebDAVfile***
- ***WebDAVresource***

---

14. Standard Library of OIL2-Callable Functions

OIL2 programs can conveniently obtain the declarations of all standard functions provided by the FARGOS/VISTA Object Management Environment core by including the file *$VISTA_ROOT/oil2Include/OMEcore.o2h*:

```
%include <OMEcore.o2h>
```

Some of these routines are described below. **Note:** all OIL2-callable functions can be utilized by C++ programmers; however, specialized C++-specific equivalents might be chosen as a matter of preference.

Language Support

int allow(string methodName)

By default, only one method can be active upon an object at a time. In some situations, it is desirable or necessary to permit a second method to be invoked upon the object despite the first method still being active. A programmer calls the **allow()** function and passes as its argument the name of the method that can be invoked.

Note: used in this way, an **allow()** permits only one invocation of the indicated method. Typically, before the method in question exits, another **allow()** call is made to permit a subsequent invocation of the method.

Also note: the FARGOS/VISTA runtime permits the allowed method to be restricted to a particular class, thus overriding the normal resolution mechanism in which an implementation in a derived class will have precedence over that of a base class if two methods have identical prototypes. None of this advanced functionality is exposed by the description above.

int alwaysAllow(string methodName)

The **alwaysAllow()** function is almost identical to **allow()** function, which was described above, with the notable exception that the allow is permanent instead of a one-shot affair.

int inCalledMethod()

The **inCalledMethod()** function returns 1 if the method was called; otherwise it returns zero if it was invoked as a new thread of execution.

int oidIsExternal(oid obj)

The **oidIsExternal()** function returns 0 if the object is resident within the local environment; otherwise it returns 1 to indicate the object is external.

nlm createNLM(string catalogName, any messageID, string defaultMessage, any parameters, any annotations)

FARGOS/VISTA's intrinsic support for Native Language Messages is a powerful feature that should be used by any programmer that expects his program to be utilized in other countries. The *catalogName* argument specifies the name of the message catalog, *messageID* identifies the message and can be an integer (X/Open style) or a string. The defaultMessage is used if a message cannot be obtained from a catalog on the target system. The *parameters* argument is an array of data that can be used to provide information that will be displayed in positional fields. The *annotations* argument provides the opportunity for including additional semantic information.

any makeUnique(any referenceCountedType)

Returns a unique copy of the passed argument. If the original argument was a reference counted structure with more than one reference, the result is a duplicate in full.

int typeOf(any)

The **typeOf()** function returns the type of the passed argument. The result is an integer whose value indicates the type. OIL2 allows the reserved keywords to be used as integer constants whose values correspond to each type. This is illustrated in the following example:

```
if (typeOf(argument) == string) {
   display("It is a string\n");
} else if (typeOf(argument) == int) {
   display("It is an integer\n");
}
```

Array and Set Manipulation

int elementCount(any complexType)

The **elementCount()** function returns the number of elements in a sparse array, associative array or set. If an invalid type is passed, a value of zero is return.

int indexExists(any sparseOrAssocArray, any integerOrStringIndex)

The **indexExists()** function is used with sparse and associative arrays to determine if a particular array element is defined. Because references to such arrays causes a new element (whose value is **nil**) to be implicitly created as a side-effect, the **indexExists()** function is often used to make this determination. Finding the first element in a sparse array serves as one example:

```
if (indexExists(a, 0) == 1) j = 0;
else j = nextIndex(a, 0);
```

Another example is to check to see if an entry exists in a table:

```
if (indexExists(table, key) == 0) return (nil);  // doesn't exist
result = table[key];
return (result);
```

int nextIndex(any sparseOrAssocArray, int subscript)

Because OIL2 arrays have the inherent ability to be sparse, the ability to discover the various subscripts is needed. An equivalent issue arises with associative arrays. The **nextIndex()** function provides this facility. For a given subscript, it returns the value of the next subscript. If there is no subscript, its return value is zero. Because associative arrays are guaranteed never to have integer subscripts with a value of zero, the code to iterate over all of the elements is straightforward:

```
for(j=nextIndex(a, 0);j != 0;j = nextIndex(a, j)) {
   // do something with element subscripted by j;
}
```

Sparse arrays that have the potential of having an entry at subscript 0 need to make an explicit check with the **indexExists()** function to see if this was the case. This makes the code somewhat more complex:

```
if (indexExists(a, 0) == 1) j = 0;
else j = nextIndex(a, 0);
if (indexExists(a, j) != 0) {
   while (j != 0) {
         // do something with index j
         j = nextIndex(a, j);
         if (j == 0) break;
   }
}
```

string getKeyForIndex(assoc assocArray, int subscript)

When iterating through an associative array using the **nextIndex()** function, the actual string key associated with the subscript is often required. This is obtained using the **getKeyForIndex()** function. For example:

```
assoc          a;
int            j, n;
string   k;

a["key value"] = 123;
j = nextIndex(a, 0);
k =  getKeyForIndex(a, j);
n = a[k]
// k is now "key value" and n is 123
```

int deleteIndex(any sparseOrAssocArray, any integerOrStringSubscript)

At some point in time, it may be necessary to remove an element that was placed in a sparse or associative array. This is performed by the **deleteIndex()** function.

Note: the sparse or associative array is passed by value, so to delete an element and update an array, the following usage is required:

```
theArray = deleteIndex(theArray, subscript);
```

set arrayToSet(array sparseArray)

This convenience function converts a sparse array to a set. The effect is to preserve the order of the elements in the array but throw away the corresponding integer subscripts.

array setToArrray(set listOfElements)

This convenience function converts a set into a dense array. The first element of the resulting array is placed in subscript 0.

any mergeArrays(…)

The **mergeArrays()** function can operate against a set of either sparse arrays or associative arrays. It returns a sparse or associative array that represents a merge of all of the arrays passed as arguments. For a given subscript, precedence is given to the last elements in the parameter list.

array orderSubscripts(any vector, int sortMode)

The **orderSubscripts()** function returns an array of subscripts that have been in ordered in such a fashion so as to permit the elements of *vector* to be accessed in a desired order. The *sortMode* flag can have several settings. For sparse and associative arrays:

- 0 = ascending by value of element
- 1 = descending by value of element
- 4 = ascending by value of element, case-insensitive comparison
- 5 = descending by value of element, case-insensitive comparison

For associative arrays only:

- 2 = ascending by value subscript key
- 3 = descending by value of subscript key
- 6 = ascending by value of subscript key, case-insensitive comparison
- 7 = descending by value of subscript key, case-insensitive comparison

Note: Whenever possible, programmers should always use the **orderSubscripts()** function rather than **sortArray()** because it eliminates the need to create a new array and potentially duplicate a large amount of data in the process.

Table 3 Predefined constants for **orderSubscripts()** in *OMEcore.o2h*

Constant Name	Function
FLAG_SORT_ASCENDING	Sort order is ascending
FLAG_SORT_DESCENDING	Sort order is descending
FLAG_SORT_BY_KEY	For associative arrays, sort by the values of the subscript key, not the values of the elements
FLAG_SORT_CASE_INSENSITIVE	Ignore case when comparing sort keys

array sortArray(any vector, int sortMode)

The **sortArray()** function is a convenience function that uses **orderSubscripts()**. It returns a new version of *vector* that is reordered to correspond to the order selected by *sortMode*.

Note: Programmers should consider using the **orderSubscripts()** function rather than **sortArray()** whenever possible because it eliminates the need to create a new array and potentially duplicate data.

String Manipulation

string charToString(int char)

A character constant or previously extracted character (e.g., from **midchar()**) can be converted to a string by the **charToString()** function.

any stringToNumber(string s, int desiredOutputType)

Sometimes a program has a text string that needs to be converted to a numeric value. The **stringToNumber()** function can be used to perform this task. The source string is passed as the first argument and the desired type of the result is passed as the second (e.g., the keyword **int** or **float** or **any**). If the desired type is not specified as **any**, the value will be coerced if necessary. Some examples:

```
int      j;
float    f;
any      v;
j = stringToNumber("123.45", int);  // j = 123
f = stringToNumber("123", float);  // f = 123.0
v = stringToNumber("123", any); // v = 123 (an int)
v = stringToNumber("123.45", any); // v = 123.45
```

array tokenizeString(string source, string delimiters, int convert)

This useful function takes a source string and parses it into tokens. The returned result is an array of tokens and the first element is in subscript 0. A token is separated from its fellows by a delimiter character; the set of possible delimiter characters is passed as the second argument. This routine can also convert tokens that appear to be numbers to the appropriate representation (e.g., an decimal or hexadecimal integer or floating point number). This conversion is requested by passing a value of one (1) as the third argument. If a value of zero (0) is passed, then such tokens will be returned as a string. The following example parses a sentence with words separated by white space. White space in this example was specified to be a space or tab:

```
words = tokenizeString(sentence, " \t", 0);
```

As another example, the following parses a comma-separated value list (e.g., something that might have been created by saving a Microsoft Excel spreadsheet) and converts the numeric fields as needed:

```
data = tokenizeString(line, ",", 1);
```

Note: double-quotes (") can be used within the *source* string to enclose elements that would have otherwise been broken out as individual tokens or converted to a number.

int length(string)

The **length()** function returns the length of a string. If passed invalid data, the value zero is returned.

int calculateStringLength(…)

The function **calculateStringLength()** computes the total length of a set of strings. If any non-string argument is passed, the value of –1 will be returned.

int findSubstring(string sourceString, string subString)

int findLastSubstring(string sourceString, string subString)

The **findSubstring()** and **findLastSubstring()** functions search through the source string, which is passed as the first argument, for the first (or last, respectively) occurrence of the indicated sub-string. If the sub-string is not found, then the value of -1 is returned.[4] If the sub-string is found within the source string, the offset within the source string of the start of sub-string is returned. The first position of the string is offset 0.

string convertCase(string sourceString, int toLowerCase)

On occasion, it can be useful to convert the letters in a string to a common case. The **convertCase()** function can be used to perform this task. If the second argument. *toLowerCase*, is the integer value 1, then all uppercase characters are converted to lowercase; the inverse conversion from lowercase to uppercase can be selected by passing a value of zero. It is also possible to select a hybrid conversion that capitalize the start of each word and convert the remaining characters to lowercase. This is selected by passing a value of 2 for the *toLowerCase* argument.

Table 4 Predefined constants for **convertCase()** in *OMEcore.o2h*

Constant Name	Function
FLAG_CONVERT_TO_UPPERCASE	All letters to uppercase
FLAG_CONVERT_TO_LOWERCASE	All letters to lowercase
FLAG_CONVERT_TO_MIXED_CASE	First letter of each word capitalized, all other letters in lowercase

int midchar(string s, int offset)

The **midchar()** function returns the character at the indicated offset in the string. The first position of the string is offset 0. If the offset represents a point outside the string, then **nil** is returned. The **charToString()** function performs the inverse operation.

string midstr(string source, int startOffset, int fragmentLength)

When working with strings, it is often necessary to extract a portion of a string. The **midstr()** function provides this capability. The desired portion is indicated by the offset of its first character within the source string and the length of the fragment. The first character of the source string is at offset 0.

string makeAsString(...)

OIL2 allows strings to be concatenated together using the addition operator (+). The **makeAsString()** function is used in a fashion similar to string concatenation,

[4] The underlying implementation of **findSubstring()** in the FARGOS/VISTA runtime is a best-of-breed algorithm that takes advantage of several optimizations that exploit native CPU instructions.

but with a few important differences. It takes arguments of any type, thus it can be used to convert integers and floating point numbers to displayable strings. It also creates a single result string, regardless of how many arguments are passed. In this way, it is more efficient that using the addition operator, which creates an intermediate result for each addition. An example:

```
result = makeAsString("count=", count, " firstName=", first, " lastName=", last,
"\n");
```

string reverseString(string data)

The characters of a string can be reversed by **reverseString()**. For strings holding binary data or encoded using a single byte character set, this is a simple inversion: if the length of the string is *n* bytes, then byte 0 of the original will be swapped with byte *n* – 1, byte 1 will be swapped with position *n* – 2, etc. The processing of a string encoded using a multi-byte character set is much more complex because the order of the bytes comprising an individual multi-byte character cannot be altered.

string stripHTML(string data)

HTML/XML directives can be stripped from a string using the **stripHTML()** function. Applications that want to ensure that unexpected HTML directives (a typical hacker ploy when attempting cross-site scripting) have not been embedded within a user-provided string can use **stripHTML()** to remove any such content. If no such directives exist, the string is returned as-is, thus no duplication is performed—this makes the function very efficient in the normal case.

string substituteText(string original, assoc searchAndReplaceValues)

For applications that need to perform a global search-and-replace operation on a string, the **substituteText()** function is a powerful utility. Its second argument is an associative array that represents a collection of text patterns that should be located and the corresponding replacement text. The subscript of each element of the associative array is the pattern to be found and the value of the element is the replacement text. An example:

```
assoc    replaceText;
string   source, result;
source = "First name=$1 Last name=$2";
replaceText["$1"] = "John";
replaceText["$2"] = "Doe";
result = substituteText(source, replaceText);
// result now = "First name=John Last name=Doe"
```

The underlying implementation of **substituteText()** handles multiple replacement text in an optimal fashion. Rather than make several calls to **substituteText()** using individual replacement patterns, it is much more efficient to make a single call that provides all of the replacement patterns in the *searchAndReplaceValues* associative array.

string substituteEnvironmentVariables(string sourceString)

The **substituteEnvironmentVariables()** function replaces references to environment variables with their corresponding values. Following long-standing Unix tradition, an environment variable's name is introduced by the character "$".

This function provides a very convenient mechanism for applications to utilize environment variables in configuration data:

```
    data = send "readLine" to bfr;
    line = substituteEnvironmentVariables(data);
```

any parseHTTPformData(string urlEncodedData, int desiredType)

Many HTTP-based requests include data that is encoding using the MIME type "*application/x-www-form-urlencoded*" (see RFC 1738). The **parseHTTPformData()** function parses such data and converts it into either an array or associative array. If maintaining the order of the elements is significant, then *desiredType* should be specified as **array**. The resulting array will start at subscript 0 and each element will be an associative array, containing a single keyword/value pair. If the order of the parameter elements is not significant, it is far more convenient to specify *desiredType* as **assoc** and obtain a single associative array that holds all of the keyword/value pairs.

```
    assocA = parseHTTPformData(urlData, assoc);
```

Note: the **parseHTTPformData()** function recognizes both "&" and ";" as end-of-field markers. Hexadecimal escapes (%xx) and plus-as-space conversions are performed as needed.

array parseHTTPuriData(string uri, string defaultScheme, string defaultAuthority, string defaultContext)

An HTTP-based application may have occasion to interpret Uniform Resource Identifiers (URIs). The syntax is specified in RFC 2396. While a fully qualified, absolute URI is relatively straightforward to decipher, the use of relative URIs is an extremely valuable tool for content creators and thus are actually more prevalent. The **parseHTTPuriData()** function returns a 5-element array that represents the parse of the *uri* string.

0. scheme (e.g., "http", "file")
1. authority
2. path
3. query
4. fragment

array parsePathComponents(string pathName)

A path name can be broken into its component elements by the **parsePathComponents()** function. If the path is absolute, subscript 0 will be filled in with either “/” or the drive prefix (e.g., “C:\”). Relative path names are parsed such that the first element of the result array is in subscript 1.

string pathComponentsToString(array components, int start, int end)

Closely related to **parsePathComponents()**, **pathComponentsToString()** creates a new path name by combining the elements of the *components* array starting at the subscript indicated by *start* and ending at the subscript indicated by *end*.

string relativePathToAbsolute(string rootDir, string relativePath)

Many relative path specifications include the use of “..” to traverse parent directories. The **relativePathToAbsolute()** function safely processes such relative paths in relation to a given root directory. The result string is an absolute path that is guaranteed to be rooted at the directory specified by *rootDir*.

Bit Manipulation

Several useful functions are available to significantly ease the complexity of bit manipulation. The **concatBits()** routine is especially useful when dealing with the bit-centric nature of a dense encoding scheme such as Macromedia's Shockwave Flash file format.

These routines operate against an idealized model of a bit field. The rightmost, least significant bit is identified as bit position 1. This is in contrast to a popular convention that labels that bit position as 0 and consequently permits easy computation of the value of each bit position: position *n* has value 2^n. The use of 1 to identify the least significant bit position only complicates the computation slightly: bit position *n* has a value of 2^{n-1}. The benefit is that a value of 0 can be used by some routines to indicate that no bit was set.

Bit field data is most frequently obtained from an integer, but integers can only hold a limited number of bits. For example, by definition, a 32-bit integer is limited to a maximum of 32-bits. Arbitrarily long bit fields can be manipulated using strings. When bit fields are provided as integers, issues related to an underlying system's storage layout byte order are automatically handled.

int lowBit(int integerValue)

The **lowBit()** function returns the lowermost bit that is set in an integer value. Bits are numbered from 1, thus the high-order bit of a 32-bit integer would be 32. If no bits are set, the value 0 is returned. For example, the lowermost bit of the value 136 would be identified as 4.

int highBit(int integerValue)

The **highBit()** function returns the uppermost bit that is set in an integer value. Bits are numbered from 1, thus the high-order bit of a 32-bit integer would be 32. If no bits are set, the value 0 is returned. For example, the uppermost bit of the value 136 would be identified as 8.

int mergeBits(int src, int srcOffset, int srcLen, int mergeInto, int mergeOffset)

The **mergeBits()** function copies from the integer value *src* a contiguous sequence of *srcLen* bits from bit positions *srcOffset* through bit position *srcOffset* + *srcLen* – 1. The returned result value is a copy of the *mergeInto* argument that has been modified by replacing bit positions *mergeOffset* through *mergeOffset* + *srcLen* – 1 with the bit sequence copied from the *src* argument. The **mergeBits()** function can be used to perform many bit manipulation operations that traditionally are achieved through a combination of masking and bi-directional shifting.

array concatBits(any sourceBitData, int bitsToAdd, any bitState)

The **concatBits()** function is used to build up a sequence of arbitrarily long bit fields. The bit field to be added is extracted from the *sourceData* as a contiguous sequence of bits starting at bit position *bitsToAdd* and onwards to the end at bit position 1. The *bitState* argument is normally passed the result from a prior call to **concatBits()**, but is specified as **nil** for an initial invocation.

The result from **concatBits()** is a two-element array. The first element (at subscript 0) is a string that contains the concatenated bit fields. If needed, the string is padded on the right with 0 bits to fill out the last byte. The second element of the array (subscript 1) is an integer that indicates the number of valid bits present. A value of 21 would correspond to a 4 byte string, whose last byte (offset + 3) would have valid data in bit positions 8 through 4 and 0 bits in positions 3, 2 and 1.

The *sourceBitData* argument is usually provided as an integer, but it may also be specified as a string. Issues related to native byte order storage layout for integers are automatically handled when an integer is passed, but strings are accessed as-is. Bit position 1 of a string occurs in the last byte of the string, thus a 10-byte long string will have bit position 1 occur in offset +9 and bit 80 (the uppermost bit) will occur in offset +0.

Debugging

int display(...)

int debugDisplay(int debugLogLevel,…)

These functions write the data passed to standard out. The **debugDisplay()** routine only outputs the data if the indicated log level is selected in the *OMElogLevel* mask.

The number of elements displayed is returned; if **debugDisplay()** does not output any data because the appropriate debug log flag was not set, then 0 is returned to indicate no data was output.

Table 5 Predefined constants for **debugDisplay()** in *OMEcore.o2h*

Constant Name	Function
debugLogLevel0	Log level 0
debugLogLevel1	Log level 1
debugLogLevel2	Log level 2
debugLogLevel3	Log level 3

int displayAsHex(...)

int debugDisplayAsHex(int debugLogLevel, …)

Similar in behavior to **makeAsHexString()**, writes any passed string data as hexadecimal characters to standard out. The **debugDisplayAsHex()** routine only outputs the data if the indicated log level is selected in the *OMElogLevel* mask.

The number of elements displayed is returned; if **debugDisplayAsHex()** does not output any data because the appropriate debug log flag was not set, then 0 is returned to indicate no data was output.

any displayVisible(string)

Displays on standard output only the alphanumeric characters in the string, stripping out any non-displayable (e.g., control) characters.

assoc getSystemInfo()

A significant number of profiling counters and configuration parameters are made available by instances of the C++ class **OMEprofileCounter**. A set of standard attributes is defined in the section below entitled "System Information Attributes". A snapshot of the entire current collection of counters and attributes can be retrieved by the **getSystemInfo()** function.

any getSystemInfoAttribute(string attributeName)

The **getSystemInfoAttribute()** function is similar to **getSystemInfo()**, except that it returns the value of the single attribute identified by *attributeName*.

Data Encoding

Data encoding routines transfer data into an alternate format that is more appropriate for the application's need. Almost without exception, these transformations are reversible and do not lose information in the process, but the size of the converted data will be different from that of the original. While encryption functions have similar attributes, they are used to hide information rather than make it more readily manipulated.

string asciiToBase64(string data, int breakIntoLines)

Binary data is often encoded for transport over 7-bit data streams (e.g., Simple Mail Transfer Protocol) using the base 64 encoding, which converts every 3 input bytes into 4 output bytes. The function **asciiToBase64()** converts an arbitrary string of data to a base 64 encoding. If the *breakIntoLines* argument is not specified as 0, then the data will be automatically formatted into carriage return/line feed-terminated lines of maximum length 72. The base 64 content transfer encoding is specified in section 5.2 of RFC 1521.

string base64ToASCII(string)

The **base64ToASCII()** function converts base 64-encoded data (such as that prepared by the **asciiToBase64()** function) into its binary equivalent. The base 64 content transfer encoding is specified in section 5.2 of RFC 1521.

string compressString(string data)

The **compressString()** function compresses the passed string and returns a string which is expected to be smaller in size. The compressed data is annotated with information that indicates the size of the original string. Compressed data can be restored to its original content by the **uncompressString()** function.

string gzipString(string data)

The **gzipString()** compresses a string into a format specified by RFC 1952. The use of **compressString()** is always preferred to **gzipString()** for two reasons: it produces smaller results and is more efficient to uncompress.

string uncompressString(string data)

The **uncompressString()** function converts data that was previously generated by **compressString()** into its original form.

string gunzipString(string data)

The **gunzipString()** function restores data that was previously compressed by a gzip-equivalent (RFC 1952 compliant) application.

string makeAsHexString(…)

The **makeAsHexString()** function is almost identical to **makeAsString()**, with the exception that any string arguments are converted to hexadecimal strings. Each byte of such a string is converted to the 2 hexadecimal characters that correspond to its value. For example, the byte value 0 is converted to “00” and the byte value 10 (an ASCII new line character) is converted to “0d”.

string hexToBinary(string hexString)

The **hexToBinary()** function is the logical inverse of the **makeAsHexString()** function. It converts a hex string into the corresponding binary data.

Note: the base 64 encoding is a more efficient (a 3:4 vs. 1:2 expansion ratio) textual encoding of binary data than hexadecimal strings (see **asciiToBase64()** for details).

string safeURI(string textString)

RFC 2396 declares that URIs are not permitted to have several characters, such as delimiters like a space or a tab. The **safeURI()** function converts a text string into a string safe for use in an URI. The inverse operation is performed by the **convertURIescapes()** function.

string convertURIescapes(string textString)

Uniform Resource Identifiers are not permitted to contain several characters, most of which are used as delimiters. When a resource uses such characters, the prohibited characters must be escaped (the **safeURI()** function performs this operation). Applications that are provided raw URIs can convert any embedded escape sequences by using the **convertURIescapes()** function.

any decodeData(string)

The **decodeData()** function converts data that was previously encoded using the **encodeData()** function.

array decodeFirstElement(string data)

The **decodeFirstElement()** function is almost identical to **decodeData()**. The difference is that it returns a two-element array. The first element (at subscript 0) is the decoded data, which is the same as would have been returned by **decodeData()**. The second element of the array holds the offset in the string where the next data element would begin.
The **decodeFirstElement()** function is used in situations where several strings, each of which had been individually prepared by **encodeData()**, were concatenated.

int decodeStringAsLength(string data)

A 4-byte string previously encoded by **encodeLengthAsString()** is decoded into an integer by the **decodeStringAsLength()** function.

string encodeData(any data, int version)

Any OIL2 data element can be encoded into a string using the **encodeData()** function. The second argument specifies the encoding version to be used. There are two encoding schemes always available: version 1 is the default set of encoding routines and version 2 is a compressed version that can be used when encoding size is more important than speed. Several other encoding schemes may be available locally, but they are not guaranteed to be universally available. The **listEncodingVersions()** function can be used to obtain information about locally supported encoding versions.

string encodeLengthAsString(int val)

An integer value can be converted into a 4-byte network-byte order (MSB … LSB) string using **encodeLengthAsString()**. It can be decoded by the **decodeStringAsLength()** function.

array listEncodingVersions()

The list of available encoding versions can be obtained by the **listEncodingVersions()** function.

Encryption

Encryption involves the scrambling of data so that it is not readable by third parties. Decryption is the inverse process that restores the jumbled data to its original state. For performance, the standard FARGOS/VISTA encryption facilities use symmetric (or shared) key algorithms, which means that the same key is used to encrypt the data as is used to decrypt it. Since both the sending and receiving party need to have the same key in order to successfully exchange encrypted data, an interrelated issue involves the exchange of the shared, yet secret, key without having its value disclosed to a third party. This key exchange is made possible using a public key exchange algorithm. Public key encryption algorithms break a key into two pieces: a private part, which must remain secret, and a public part, which can be disclosed to third parties without compromising the system.

array makePublicKeyPair(string secret)

The **makePublicKey()** function returns a two-element array that represents a public key pair:

0	private key
1	public key

The string *secret* is used to generate a unique public key pair.

array makeSessionKey(string publicKey, string randomData)

For performance, bulk data is encrypted using a symmetric encryption algorithm. For security, the key to be used should be randomly chosen. The **makeSessionKey()** function returns a two-element array:

0. encrypted session key
1. session key

The *publicKey* argument is the value of the public key (subscript 1 of the result array) obtained from a prior call to **makePublicKeyPair()**. The randomness of the generated key is obtained from the *randomData* argument. It should be a 128-bit (16 byte) string. The **makeRandomKey()** function is the preferred mechanism to generate this data, since it will automatically take advantage of hardware support for random number generation.

The session key (subscript 1) should be used as the shared key for a symmetric encryption algorithm (i.e., the secret for passed in call to the **initializeCipher()** function). The encrypted session key (subscript 0) can be forwarded across the network and decoded using the **decryptSessionKey()** function.

string decryptSessionKey(string localPrivateKey, string encryptedKey)

A session key previously encrypted by **makeSessionKey()** is decrypted using the **decryptSessionKey()** function. The *encryptedKey* holds the data that was the first element (subscript 0) of a prior **makeSessionKey()** call by a remote host. The *localPrivateKey* is the private key (subscript 0) from **makePublicKeyPair()** call.

array initializeCipher(string secret, int dir, string initVector)

Before a cipher can be used, it must be initialized. The **initializeCipher()** function performs the initialization. On some platforms with support for cryptographic hardware, this will set up the hardware device. The argument *secret* is the key used to perform encryption or decryption. The *dir* argument specifies what operation is being performed. Two values are permitted:

Constant Name	Value	Function
FLAG_ENCRYPT	1	encrypt
FLAG_DECRYPT	2	decrypt

The *initVector* argument holds an initialization vector for the cipher. Initialization vectors permit additional permutations of an encrypted result: for a given item of data and secret key, different initialization vectors will yield different encryption results. The same initialization vector must be used for decryption. The initialization vector must be a 32-byte string of displayable hexadecimal characters (0 – 9, a – f). The 32-byte string of hexadecimal characters will be converted to a 16-byte (or 128-bit) vector. The design of the initialization vector derives from the specification of the Advanced Encryption Standard. Although OIL2 has no trouble manipulating binary strings, many other implementation languages do not have the same capabilities. Thus, two advantages of this representation are that regardless of the character set of the source file (e.g., EBCDIC), the same initialization vector will be obtained and it is easy to represent binary information.

After a cipher has been initialized, depending on the direction indicated by the *dir* parameter, the respective **encryptMessage()** or **decryptMessage()** function can be used repeatedly. When a cipher is no longer needed, it should be freed using the **freeCipher()** function. The return result from **initializeCipher()** is an array, but it should be viewed as an opaque data structure.

int freeCipher(array cipherID)

The **freeCipher()** function frees all data structures (and any cryptographic hardware resources) associated with a cipher that was previously allocated by a **initializeCipher()** call.

string decryptMessage(array cipherData, string message)

The **decryptMessage()** function decrypts data previously generated by a call to **encryptMessage()** (potentially, on a different host or at some previous point in time) using a previously initialized cipher. The argument *cipherData* is obtained from a call to **initializeCipher()**.

string encryptMessage(array cipherData, string message)

The **encryptMessage()** function encrypts an arbitrary string of data using a previously initialized cipher. The data, *message*, will automatically be logically padded as needed to end on a 16-byte block boundary. The required number of blocks will automatically be encrypted to traverse the entire length of the data and any padding to needed to end on a 16-byte block boundary will be performed. The original *message* string remains unmodified and the pad operation may copy a maximum of 15 bytes of data, thus programmers need not be concerned about the size of the message or performing padding on their own. The argument *cipherData* is obtained from a call to **initializeCipher()**.

string makeRandomKey(int bits)

The **makeRandomKey()** function can be used to generate several random bytes of data. If the underlying native host operating system has hardware support for random number generation, it is automatically used.

int getRandomInteger(int upperBound)

Returns a random 32-bit integer between 0 and the passed *upperBound* argument. On platforms that support it, the random integer is obtained from a random number generator device.

Note: the **makeRandomKey()** function generates a random set of bytes as a string.

string SHA1hash(string data)

The **SHA1hash()** function performs a Secure Hash Algorithm 1 over the passed data string and returns a 160-bit (20 byte) string. See Federal Information Processing Standard 180-1 for the specification.

string MD5hash(string data)

The **MD5hash()** function computes a message digest over the passed data string and is used by several Internet RFCs. See RFC 1321 for the specification of the algorithm.

File System Information

assoc getFileInfo(string fileName)

Returns an associative array of attributes that describe the selected file; **nil** is returned if there is an error.

- type
- fileName
- bytes
- lastModifiedLocalTime
- lastModified

Known types include:

- regularFile
- symbolicLink
- directory
- characterDevice
- blockDevice
- fifo
- socket

array listDirectory(string directoryName)

The **listDirectory()** function returns a list of the files in the specified directory; **nil** is returned if the directory does not exist.

int makeDirectory(string directoryName)

A new directory can be made in the local file system using the **makeDirectory()** function. A return value of zero indicates success.

int removeDirectory(string directoryName)

An existing empty directory in a local file system can be removed with the **removeDirectory()** function. A return value of zero indicates success.

int renameFile(string orgFileName, string newFileName)

An existing local file can be renamed with the **renameFile()** function. A return value of zero indicates success.

int unlinkFile(string fileName)

The directory entry of an existing file can be removed by using the **unlinkFile()** function. A return value of zero indicates success.

Time Manipulation

int timeDifference(assoc absoluteTime1, assoc absoluteTime2)

Returns the difference in seconds between two absolute times.

assoc convertLocalRelativeTimeToAbsolute(int relativeTime, int toGMT)

On a given system, a local relative time (obtained from a call on the local system to **getLocalRelativeTime()**) can be converted to an absolute time with this function. A relative time value obtained by a call to **getLocalRelativeTime()** on a remote system will have undefined semantics within the local system. Local interpretation of such a remote value should be viewed as invalid. In contrast, an absolute time can be transferred between systems without altering its meaning. An absolute time can be relative to the local time zone or GMT. If the second argument, *toGMT*, is non-zero, the resulting time is relative to GMT instead of the local time zone.

The absolute time is represented as an associative array whose subscript keys are:

- seconds (0 – 61, to handle leap seconds)
- minutes (0 – 59)
- hours (0 – 23)
- dayOfMonth (1 – 31)
- month (1 = January, 2 = February, etc.)
- year
- dayOfWeek (0=Sunday, 1 = Monday, etc.)
- dayOfYear (0 – 365)
- isDST (Boolean)
- gmtDelta (in seconds)
- localTimezoneName (string)

Note: on most systems, the TZ environment variable influences the time zone name. Because of the inherent variability of time zone names, the value of *gmtDelta* provides the definitive identification of time zone.

assoc convertRFC1123date(string)

This function is the counterpart to **rfc1123Date()** and converts strings that conform to the date specification in RFC 1123 to an absolute time. This routine does not assume that the input string strictly conforms to the RFC 1123 specification, so it is year 10000 compliant. It will tolerate leading white space, day names longer than 3 characters, missing commas, month names longer than 3 characters, and extra white space between fields.

int getLocalRelativeTime()

Returns the current local relative time as a 32-bit integer. The value has a granularity of seconds but is only valid within the confines of the local system. Due to its compact size, it is useful in many situations in which a program is only interested in a measure of elapsed time. It must be converted to an absolute time by the **convertLocalRelativeTimeToAbsolute()** function before it can be interpreted as a time/date value that can be have meaning to another system.

int getRelativeMilliseconds()

For timing purposes, the **getRelativeMilliseconds()** function provides a finer granularity measurement of time than **getLocalRelativeTime()**. The value returned by **getRelativeMilliseconds()** is not interpreted directly. A measurement of elapsed time in milliseconds is obtained by calculating the difference in the values returned by two calls to **getRelativeMilliseconds()**.

Note: an alternative that also provides millisecond-level granularity is to retrieve the value of the *millisecondsUp* system information attribute:

```
ms = getSystemInfoAttribute(“millisecondsUp”);
```

The retrieved value for *millisecondsUp* indicates the number of milliseconds elapsed since the FARGOS/VISTA-based application was initialized, thus it always starts with an initial value of 0. There is no mechanism provided to convert a *millisecondsUp* value to an absolute time reference.

string iso8601Date(assoc absoluteTime)

The **iso8601Date()** function is similar to the **rfc1123Date()** function, but generates a different output format (ISO 8601) that is used by WebDAV-enabled applications (see Appendix 2 of RFC 2518). Such dates appear similar to:

yyyy-mm-ddThh:mm:ss+tzOffset

string rfc1123Date(assoc absoluteTime)

Converts an absolute time to a string that conforms to RFC 1123. In short, a string that looks like:

Day, dd Mon yyyy hh:mm:ss TZN

where:

- Day is a 3 character abbreviation of the day of the week (e.g., Sun, Mon, Tue, etc.)
- dd is a 2 character day of the month (e.g., 01 - 31)
- Mon is a 3 character abbreviation of the month name (e.g., Jan, Feb, Mar, etc.)
- yyyy is a 4 character representation of the year (e.g., 2000).
- hh is a 2 character representation of the hour (e.g, 00, 01, …, 13, 23)
- mm is a 2 character representation of the minute (e.g., 00 - 59)
- ss is a 2 character representation of the seconds (e.g., 00 - 59)

The benefit of this format is that each field appears at a fixed position; however, programmers that exploit this will end up with a year 10000 problem.

Note: on most systems, the TZ environment variable influences the time zone name.

Access Control Lists

All objects within a FARGOS/VISTA environment have a distinct access control list associated with them. Whenever an object is created, it must be provided with an initial access control list.

assoc makeDefaultACL()

The **makeDefaultACL()** function is the most frequently used access control list-related function. It permits the creator of the object full access and disallows all others.

assoc addUserToACL(assoc existingACL, string userInfo, assoc permittedMethods)

An ACL can be extended using the **addUserToACL()** function.

assoc makePermitEveryoneACL()

The **makePermitEveryoneACL()** function creates an ACL that permits the owner of the object and all others full access.

assoc createACLthatAllowsOthers(...)

Occasionally, the default ACL prepared by **makeDefaultACL()** is too restrictive because it does not permit any user other than the owner of an object to access it. The function **createACLthatAllowsOthers()** creates an ACL that allows other users to access only the methods that were explicitly specified as arguments to the function. If no arguments are provided, it is effectively the equivalent of **makeDefaultACL()**.

assoc createACLthatDisallowsOthers(...)

Occasionally, the default ACL prepared by **makeDefaultACL()** is much too restrictive because it does not permit any user other than the owner of an object to access it. The function **createACLthatDisallowsOthers()** creates an ACL that allows other users to access any of an object's methods except for the methods explicitly specified as arguments to the function. If total access to an object is desired, then no arguments would be passed and this is effectively the equivalent of **makePermitEveryoneACL()**. Some examples:

```
fullAccessACL = createACLthatDisallowsOthers();
dontAllowDeleteACL = createACLthatDisallowsOthers("deleteYourself", "delete");
```

oid createNewOIDthatOnlyAllowsOthers(oid obj, array permittedMethods)

An object has no control over its initial access control list as that was prepared by the creator of the object. Usually, this is not an issue; however, objects that provide services can find it useful to have special access control lists that only permit the invocation of a few methods of an object. The **createNewOIDthatOnlyAllowsOthers()** function provides a means by which an object can hand out references to itself that restrict what methods users can invoke against it. The first argument is an object ID that refers to the object and the second is an array of permitted method names. The resulting object ID will only permit the invocation of methods whose names were present in the array.

User Authentication

int becomeUser(string userName, string password)

The user associated with a thread can be set with the **becomeUser()** function. It takes two arguments, the user's login name and the associated authentication data, which is typically a password. There are 3 possible return values:

- -1 – the user is unknown
- 0 – although the user is recognized, the provided *password* data did not enable the user to be authenticated
- 1 – the user is known and was successfully authenticated

string becomePseudoUser()

A unique, anonymous pseudo user can be set as the user associated with a thread. This is useful for service providers that need to perform operations on behalf of unauthenticated users and wish to prevent granting access to objects that would normally be accessible by the service provider but should not be accessible by anonymous users. The name of the new pseudo user is returned.

Services

int registerService(string serviceName, oid obj, int exportable)

Registers the indicated object as a named service. Named services can be used as the targets of OIL2 **send** statements. The object Id of a named service can also be retrieved using the **lookupLocalService()** function. The third argument indicates if the service should be made known to other systems participating in the FARGOS/VISTA-based infrastructure.

int unregisterService(string serviceName, oid obj)

Unregisters a named service that was previously registered by a **registerService()** call.

oid lookupLocalService(string serviceName)

Returns the object Id of a named service; **nil** if there is no such corresponding object. The OIL2 **send** statement permits the use of a named service as a destination, so this routine is not always needed. It can be useful for performance reasons; for example, the lookup of the service can be done once outside the body of a loop.

Note: object Ids are valid anywhere in the FARGOS/VISTA infrastructure, but a named service often is only registered locally. The **lookupLocalService()** function is a way to obtain a reference to a local service that can subsequently be passed onto another system. The **AnnounceServices** class can export such services to peer systems.

assoc listRegisteredServices()

Returns a list of the names of all locally registered services. Each entry is subscripted by the name of the service; the value of the entry is the object Id of the service provider.

array listRemoteSystems()

Returns an array of object Ids corresponding to the ObjectCreator object on each known remote system.

15. System Information Attributes

A large number of runtime statistics are maintained during the operation of the system. C++ programmers can register additional instrumentation using the C++ class **OMEprofileCounter()**. Current values can be retrieved via getSystemInfo() or getSystemInfoAttribute() functions. C++ programmers can also utilize the lower-level routines such as **OMEprofileCounter::getValueOfCounter()**. Because the set of available of statistics can be extended by applications, the table below cannot reflect the entire list of statistics available.

Attribute Name	Description
IOmaxReadBuffer	Maximum number of bytes retrieved by **IOobject:readBytes** if a length was not specified.
IOmaxVectors	Maximum number of vectors that can be written by an atomic **writev()** (or equivalent) function call. **IOobject:writeVectorOfBytes** can handle an arbitrary number of vectors, but it may be required to make repeated calls to **writeVectorOfBytes()**. If *IOmaxVectors* is small (e.g., 16) and the vector to be output is composed of numerous but short strings, it can be beneficial to consolidate the vector by concatenating the strings and thus avoid many repeated calls to **writeVectorOfBytes**.
IOtotalDescriptorsCreated	Total number of I/O descriptors created (**IOobject**).
IOtotalDescriptorsDeleted	Total number of I/O descriptors deleted (**IOobject**). Total descriptors in use = *IOtotalDescriptorsCreated* – *IOtotalDescriptorsDeleted*.
IOtotalEmulatedWriteVectors	Total number of **writeVectorOfBytes** converted to a sequence of **writeBytes** calls because underlying O/S or I/O descriptor technology does not support scatter/gather operations. The optimal value is zero.
IOtotalFileReads	Total number of **read()** calls on a file (**IOobject:readBytes**).
IOtotalFileSelectRead	Total number of **selectForRead** calls on a file (**IOobject:selectForRead**).
IOtotalFileSelectWrite	Total number of **selectForWrite** calls on a file (**IOobject:selectForWrite**).
IOtotalFileWriteVectors	Total number of **writev()** (or equivalent) calls on a file (**IOobject:writeVectorOfBytes**).
IOtotalFileWrites	Total number of **write()** calls on a file (**IOobject:writeBytes**).

Attribute Name	Description
IOtotalFilesCreated	Total number of files opened.
IOtotalFilesDeleted	Total number of files closed. Number of files in use = *IOtotalFilesCreated – IOtotalFilesDeleted*.
IOtotalSocketRecvFroms	Total number of **recvfrom()** calls on a socket (**IOobject:receiveDatagram**).
IOtotalSocketRecvs	Total number of **recv()** calls on a socket (**IOobject:readBytes**).
IOtotalSocketSelectRead	Total number of **selectForRead** calls on a socket (**IOobject:selectForRead**).
IOtotalSocketSelectWrite	Total number of **selectForWrite** calls on a socket (**IOobject:selectForWrite**).
IOtotalSocketSendTos	Total number of **sendto()** calls on a socket (**IOobject:sendDatagram**).
IOtotalSocketSends	Total number of **send()** calls on a socket (**IOobject:writeBytes**).
IOtotalSocketWriteVectors	Total number of **writev()** calls on a socket (**IOobject:writeVectorOfBytes**).
IOtotalSocketsAccepted	Total number of incoming connections accepted.
IOtotalSocketsCreated	Total number of all sockets created.
IOtotalSocketsDeleted	Total number of all sockets deleted. Total sockets in use = *IOtotalSocketsCreated – IOtotalSocketsDeleted*.
IOtotalUnixSocketsCreated	Total number of Unix file domain sockets created.
IOtotalUnixSocketsDeleted	Total number of Unix file domain sockets deleted. Total Unix file domain sockets in use = *IOtotalUnixSocketsCreated – IOtotalUnixSocketsDeleted*.
IOtruncReadBuffer	Number of times an explicitly allocated **IOobject:readBytes** buffer had to be truncated because there was insufficient data available. The optimal value is zero.
IOtruncReadDatagramBuffer	Number of times an explicitly allocated **IOobject:receiveDatagram** buffer had to be truncated because there was insufficient data available. The optimal value is zero.
alternativeMethodTotal	Total number of methods that have multiple implementations (typically a result of being specified as **unique** so that the actual implementation is selected based on the arguments passed at runtime).
classTotal	Total number of classes currently loaded in the local FARGOS/VISTA Object Management Environment.
cpusAvailable	The number of CPUs administratively allocated to the FARGOS/VISTA Object Management Environment process.
hostName	The name of the local host.

Attribute Name	Description
maximumCPUs	The maximum number of CPUs the FARGOS/VISTA Object Management Environment is configured to be capable of utilizing. It does not mean that such CPUs exist. The *cpusAvailable* attribute will always be between 1 and the value of *maximumCPUs*.
methodTotal	The total number of methods currently loaded in the local FARGOS/VISTA Object Management Environment.
millisecondsUp	The total number of milliseconds that have elapsed since the FARGOS/VISTA-based process began execution.
minWorkForMultiprocessing	The minimum number of works units that must be queued before an additional CPU will be utilized. For maximum utilization of multiple CPUs, the value of *minWorkForMultiprocessing* can be set to 1; however, that overall efficiency can be impacted. It takes some work to both start and stop a CPU (perhaps inertia is a useful analogy) and simultaneous operation of multiple CPUs invariably creates contention for locks on critical global data structures. If there is insufficient work available for true parallel execution, two CPUs will interfere with each other to such an extent as to make the system run slower than if there was only a single CPU.
nameSpaceTotal	The total number of distinct name spaces defined in the local FARGOS/VISTA Object Management Environment.
processID	The process Id assigned by the native host operating system to the local FARGOS/VISTA Object Management Environment.
slicesOnKernelThread-0	Total number of time slices executed by the FARGOS/VISTA Object Management Environment scheduler utilizing the native operating system's primary kernel thread. If more than one CPU is being used (see *cpusAvailable*), there will be a corresponding *slicesOnKernelThread-N* attribute for each additional CPU 1 through (*cpusAvailable* – 1).
stopFlag	Current run state of the FARGOS/VISTA-based application. A non-zero value means a stop has been requested.
totalArrayDeepCopies	The total number of times a sparse array had to be duplicated as a consequence of a copy-on-write operation. The optimal value is zero.
totalAssocDeepCopies	The total number of times an associative array had to be duplicated as a consequence of a copy-on-write operation. The optimal value is zero.
totalOIDsCreated	Total number of object Ids created or imported during the current execution of the FARGOS/VISTA-based application.

Attribute Name	Description
totalOIDsDeleted	Total number of object Ids deleted during the current execution of the FARGOS/VISTA-based application. The total number of objects known to the application is no more than *totalOIDsCreated – totalOIDsDeleted*; however, it might be less than that.
totalObjectsCreated	Total number of objects created or imported during the current execution of the local FARGOS/VISTA Object Management Environment process.
totalObjectsDeleted	Total number of objects deleted or removed during the current execution of the local FARGOS/VISTA Object Management Environment. The total number of objects resident can be computed as *totalObjectsCreated – totalObjectsDeleted*. Note that the total number of objects resident is not the same as the total number of objects known to the system—many other objects can be either paged out (e.g., **PersistentObject**s) or located within a remote peer.
totalSetDeepCopies	The total number of times a set had to be duplicated as a consequence of a copy-on-write operation. The optimal value is zero.
totalStringDeepCopies	The total number of times a string had to be duplicated as a consequence of a copy-on-write operation. The optimal value is zero.
totalThreadsAllowed	The total number of times the execution of a method would have been delayed if not for the fact it was permitted to proceed because the method had been previously allowed (e.g., via an **allow()** or equivalent call).
totalThreadsCreated	The total number of threads created during the execution of the local FARGOS/VISTA Object Management Environment. This has a strong correlation with the number of methods executed.
totalThreadsDelayed	The total number of times the execution of a method was delayed. This is usually because another thread was active upon the same object. The optimal value is zero.
totalThreadsDeleted	The total number of threads terminated during the execution of the local FARGOS/VISTA Object Management Environment. The total number of threads in use is equal to *totalThreadsCreated – totalThreadsDeleted*.
totalTimeEventCalls	Total number of times the timer queue was evaluated. This monotonically increasing value should be interpreted in the context of *millisecondsUp*. The efficiency of the system is proportional to *totalTimeEventCalls / millisecondsUp*.

Attribute Name	Description
totalWaitForIOcalls	Total number of times the FARGOS/VISTA-based application checked to see if pending I/O operations were ready to proceed. This monotonically increasing should be interpreted in the context of *millisecondsUp*. The efficiency of the system is proportional to *totalWaitForIOcalls* / *millisecondsUp*; smaller values are better.
vista_cpu	A string identifying the CPU architecture for which the FARGOS/VISTA-based executable was compiled (e.g., "*i386*", "*sparc*", "*i86pc*").
vista_major_version	An integer identifying the major version number of the FARGOS/VISTA release. Corresponds to field *V* in the version Id *V-N.R*.
vista_minor_version	An integer identifying the minor version number of the FARGOS/VISTA release. Corresponds to field *N* in the version Id *V-N.R*.
vista_os	A string identifying the native operating system for which the FARGOS/VISTA-based executable was compiled (e.g., "*Linux*", "*SunOS*", "*OpenBSD*")
vista_release_version	An integer identifying the release version number of the FARGOS/VISTA release. Corresponds to field *R* in the version Id *V-N.R*.