Excerpts from the

Microsoft Multimedia Operating System (MMOSA)

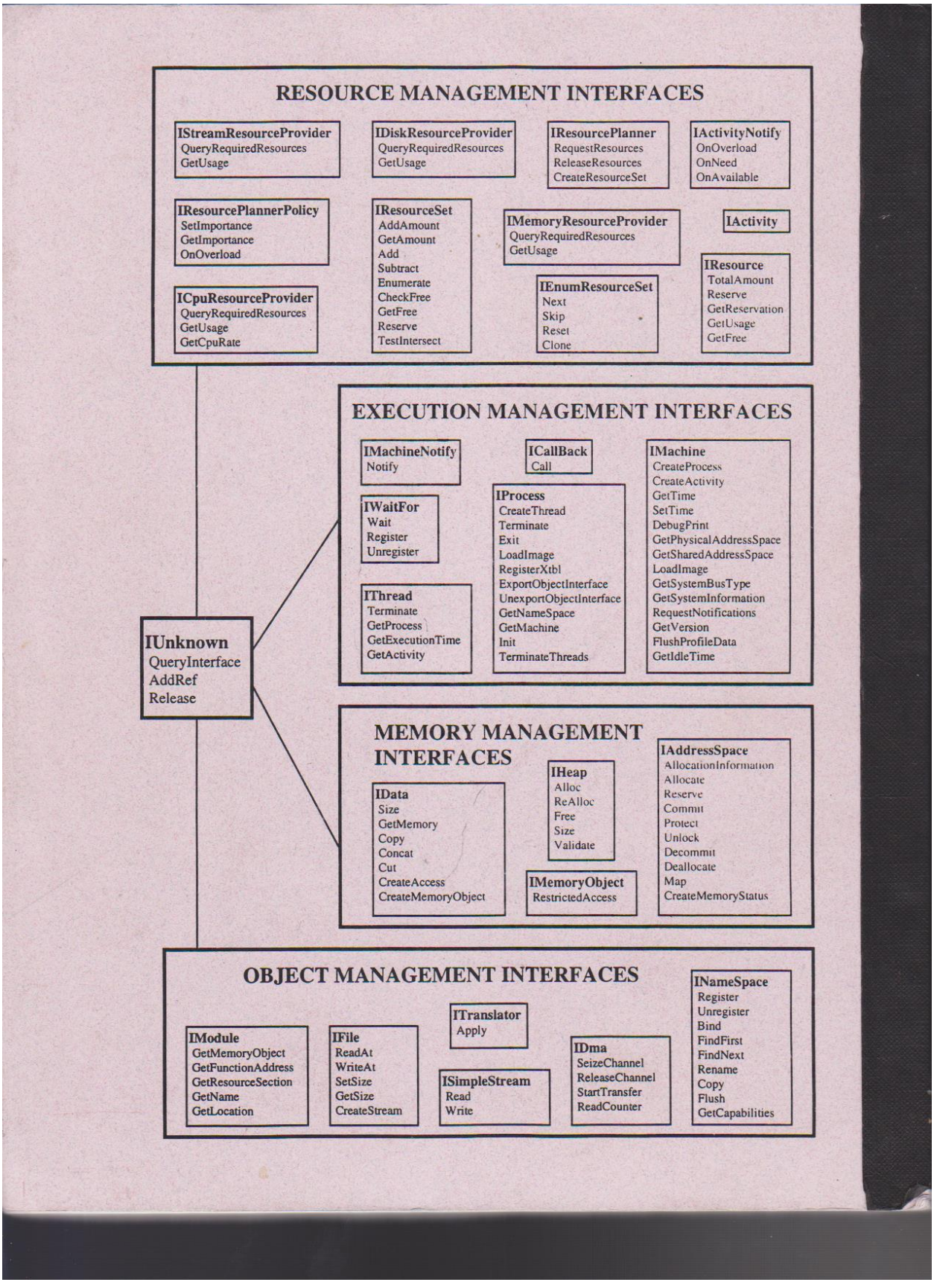
OEM Adaption Kit (OAK)

Prepared for

Microsoft Interactive Television (MITV)

June 1996

Seth McEvoy, Programmer Writer


RESOURCE MANAGEMENT INTERFACES
IStreamResourceProvider
QueryRequiredResources
GetUsage
IDiskResourceProvider
QueryRequiredResources
GetUsage
IResourcePlanner
RequestResources
ReleaseResources
CreateResourceSet
IActivityNotify
OnOverload
OnNeed
OnAvailable
IResourcePlannerPolicy
SetImportance
GetImportance
OnOverload
IResourceSet
AddAmount
GetAmount
Add
Subtract
Enumerate
CheckFree
GetFree
Reserve
TestIntersect
IMemoryResourceProvider
QueryRequiredResources
GetUsage
IActivity
IResource
TotalAmount
Reserve
GetReservation
GetUsage
GetFree
ICpuResourceProvider
QueryRequiredResources
GetUsage
GetCpuRate
IEnumResourceSet
Next
Skip
Reset
Clone
EXECUTION MANAGEMENT INTERFACES
IMachineNotify
Notify
ICallBack
Call
IMachine
CreateProcess
CreateActivity
GetTime
SetTime
DebugPrint
GetPhysicalAddressSpace
GetSharedAddressSpace
LoadImage
GetSystemBusType
GetSystemInformation
RequestNotifications
GetVersion
FlushProfileData
GetIdleTime
IWaitFor
Wait
Register
Unregister
IProcess
CreateThread
Terminate
Exit
LoadImage
RegisterXtbl
ExportObjectInterface
UnexportObjectInterface
GetNameSpace
GetMachine
Init
TerminateThreads
IThread
Terminate
GetProcess
GetExecutionTime
GetActivity
IUnknown
QueryInterface
AddRef
Release
MEMORY MANAGEMENT INTERFACES
IAddressSpace
AllocationInformation
Allocate
Reserve
Commit
Protect
Unlock
Decommit
Deallocate
Map
CreateMemoryStatus
IHeap
Alloc
ReAlloc
Free
Size
Validate
IData
Size
GetMemory
Copy
Concat
Cut
CreateAccess
CreateMemoryObject
IMemoryObject
RestrictedAccess
OBJECT MANAGEMENT INTERFACES
INameSpace
Register
Unregister
Bind
FindFirst
FindNext
Rename
Copy
Flush
GetCapabilities
ITranslator
Apply
IModule
GetMemoryObject
GetFunctionAddress
GetResourceSection
GetName
GetLocation
IFile
ReadAt
WriteAt
SetSize
GetSize
CreateStream
ISimpleStream
Read
Write
IDma
SeizeChannel
ReleaseChannel
StartTransfer
ReadCounter

GUIDE TO PROGRAMMING

The MMOSA Guide to Programming covers general architure, tutorial, and programming topics. Specific information on individual interfaces, methods, functions, data types, etc., is covered in the MMOSA Reference. Topics pertaining to device drivers are covered in the MMOSA Device Driver Kit.

Overview of MMOSA

The Multi-Media Operating System Architecture (MMOSA) is a new embedded operating system with a combination of speed, power, and simplicity that will enable software developers and hardware engineers to create a wide array of consumer products.

MMOSA has a compact high-speed design that is small enough to run tiny hand-held computers and yet it's also fast and powerful enough to run huge interactive television systems. One of MMOSA's most unique features is that it can run on many different processors, platforms, and networks using the advanced distributed object-oriented style of programming, COM (Component Object Model).

To achieve these goals, MMOSA was created to be very small and compact. It has only a few interfaces that the systems programmer needs to allocate memory, create objects and manage objects, and schedule complex tasks in a distributed multi-processing, multi-threaded environment. As a result, MMOSA can be learned quickly and easily.

MMOSA was not designed to satisfy every need nor was it designed to replace all other operating systems. But for situations where high speed and compact size are an absolute necessity, MMOSA will be the appropriate operating system.

COM for the Kernel

There are many models for object-oriented programming. The Component Object Model (COM) provides the model for programming in the kernel environment.

- Condition objects used to start and stop threads, based on a condition. For example, a thread can be put to sleep and told to wait until a interrupt signal is received. When the condition is signalled, the thread resumes execution.
- Read/write locks can also be used to provide multiple reader, single writer locking for threads in the same process. This is similar to mutexes but allows other threads to read a locked portion of memory.

NOTE Constraints, mutexes, conditions, and read/write locks are not COM objects; they are simply *built-in* objects that just contain a data structure.

Memory Objects

Memory objects allow applications in one process to share memory with applications in other processes.

[Additional information to come.]

Environment and Tools

The following topics are covered in this section.

Hardware Platforms. What hardware does the kernel run on?

Running the kernel. How to run the kernel and launch programs.

Building the kernel. How to build the kernel.

Creating applications. How to compile and link programs for the kernel.

Hardware Platforms

The kernel is designed to operate on different hardware platforms.

At present, the kernel supports the following two platforms.

Intel x86

The kernel will run on an Intel 486 and Pentium PC platform. A floppy drive, keyboard, and 640K of memory are required. 386 support is contemplated but not yet implemented. Hard drive, extended memory, expanded memory, and graphics are not required for a minimum system.

MIPS R3000

The kernel also supports the MIPS R3000 chip. However, unless otherwise noted, all examples are for the Intel platform.

MMID values are unique for all current objects, but the MMID values are recycled, so a current object may be assigned the same MMID value as a past object which doesn't exist anymore.

MMOSA Standard Library (mmstdlib)

The following functions are extra functions that are defined in MMSTDLIB.H instead of MMOSA.H:

WaitFor
CreateCallBack
CloneNameSpace
WaitOnStr
SignalStr
GetResourceAddress
LoadStringResource

MUTEX

```
typedef struct  _MUTEX
    {
    UINT _mutex_state[ 1 ];
    }   MUTEX;
```

A mutual exclusion (mutex) object for sychronizing threads. Only one thread at a time can own a mutex, which enables mutexes to be used to coordinate access to shared resources. Mutex values are as follows.

Definition	Value	Description
SIGNALLED	1	Not owned by any thread..
NONSIGNALLED	0	Owned by a thread.

Mutex_Init

void Mutex_Init(*mutex*)
MUTEX **mutex* // out

Mutex_Init creates a mutex system object and initializes a **MUTEX** structure so that it refers to this object.

Parameter

mutex
Address of the **MUTEX** structure that the function initializes.

Return value

This function has no return value.

Remarks

This method must be called before any of the other functions in this interface are used. However, calling this function by itself does not lock the mutex. You must call **Mutex_Lock** or **Mutex_TryLock** to use the mutex.

Remember to call **Mutex_Destroy** when the mutex is no longer needed.

Example

```
MUTEX mutex;    /* Previously defined mutex object. */

Mutex_Init(&mutex);
...
Mutex_Destroy(&mutex);
```

See Also

Mutex_Destroy, **Mutex_Lock**, **Mutex_TryLock**

Mutex_Lock

void Mutex_Lock(*mutex*)
MUTEX **mutex* // in, out

Mutex_Lock acquires the mutex. If the mutex is currently held by another thread, the thread waits until the mutex is available. If the mutex is already held by the calling thread, a deadlock may occur or an unspecified run-time error or an exception.

Parameter

mutex
Address of the **MUTEX** structure corresponding to the mutex to be locked.

Return value

This function has no return value. However, if the mutex is already held by the calling thread, an unspecified run-time error or exception may occur.

Remarks

If another thread has already acquired the lock on this mutex, the thread making the **Mutex_Lock** call must wait. The mutex is not freed up until the current owner of the mutex unlocks it. At that point, the thread calling **Mutex_Lock** gains ownership of the mutex and continues execution (unless another thread acquired the lock instead).

If a thread attempts to lock a mutex while already holding a lock on that same mutex, a run-time error results. If it is unknown whether the current thread owns a mutex, call **Mutex_Locked** before calling **Mutex_Lock** to avoid run-time errors.

To avoid waiting when the mutex is not available, call **Mutex_TryLock**.

When different threads are printing messages to the console, Mutex_Lock can insure that each thread's message prints to the console at the appropriate time as demonstrated in the following example.

Example

```
#define NUMTHREADS 3
MUTEX PrintMutex;
void main()
{
IThread *pThd[NUMTHREADS];
UINT NumThreads = NUMTHREADS;
UINT i;
IProcess *pPrc;
SCODE StatusCode;
void THREAD_LINKAGE PrintThread(THREAD_ARGUMENT arg);

Mutex_Init(&PrintMutex);
pPrc = CurrentProcess();

for (i = 0; i < NUMTHREADS; i++) {
    if((StatusCode = pPrc->CreateThread(PrintThread, (THREAD_ARGUMENT)i,
NULL, &pThd[i])) != 0) {
        Printf(TEXT("Couldn't create thread %d. StatusCode is %x\r\n"),
i, StatusCode);
        Exit(0);
    }
    SleepUntil(TIME_RELATIVE(TIME_MILLIS(20)));
    Mutex_Lock(&PrintMutex);
    Printf(TEXT("Main has the PrintMutex locked for printing.\r\n"));
    Printf(TEXT("Back from the CreateThread method that created thread
%d, got %x.\r\n"), i, pThd[i] );
    Printf(TEXT("Now, unlock the PrintMutex in Main.\r\n"));
    Mutex_Unlock(&PrintMutex);
}
Mutex_Lock(&PrintMutex);
Printf(TEXT("Main has the PrintMutex locked for printing.\r\n"));
Printf(TEXT("Since the last thread did not lock PrintMutex, its print
message is\r\nout of sequence.\r\n"));
Mutex_Unlock(&PrintMutex);
```

AtomicAdd

UINT AtomicAdd(*pCount*, *Plus*)
UINT **pCount* // in, out
UINT *Plus* // in

AtomicAdd atomically adds the number specified by the *Plus* parameter to the value stored in the **pCount* parameter.

Parameters

pCount
Address of a value that on entry contains the initial value that is to be increased by *Plus*, and that on return contains the new value.

Plus
Specifies the value to add to the **pCount* parameter.

Return Value

Returns the new value of **pCount*.

Remarks

AtomicAdd(*pCount*, 1) is equivalent to **AtomicInc**(*pCount*).

Example

```
UINT CountIn=0, CountOut=0;
UINT Plus = 5;

CountOut = AtomicAdd(&CountIn, Plus);
Printf(TEXT("After adding %d, the input count now is %d (changed by
            pass-by-reference).\r\n"), Plus, CountIn);
Printf(TEXT("After adding %d, the output count now is %d (function's
            return value).\r\n"), Plus, CountOut);
```

See Also

AtomicSub

AtomicMaybeAdd

AtomicCmpAndSwap

BOOL AtomicCmpAndSwap(*pTarget*, *Oldvalue*, *Newvalue*)
UINT **pTarget* // in, out
UINT *Oldvalue* // in
UINT *Newvalue* // in

AtomicCmpAndSwap atomically sets **pTarget* to *Newvalue* when the initial value of **pTarget* equals *Oldvalue*.

Parameters

pTarget
Address of a value that is compared to *Oldvalue*. When the two values are equal, the value of **pTarget* is reset to *Newvalue*. When the two values are unequal, the value of **pTarget* is not changed.

Oldvalue
Specifies the value for comparison.

Newvalue
New value that is written to **pTarget* when the initial value of **pTarget* matches *Oldvalue*.

Return Value

TRUE if a new value is set for *pTarget*, else FALSE.

Example

```
UINT CountIn=6;
UINT NewValue = 15, OldValue;
BOOL YesOrNo;

OldValue = CountIn;
YesOrNo = AtomicCmpAndSwap(&CountIn,OldValue,NewValue); /*TRUE result*/
if(YesOrNo)
    Printf(TEXT("The new value, %d, replaces the old value, %d.\r\n"),
                CountIn, OldValue);
else
    Printf(TEXT("The AtomicCmpAndSwap function did not operate as
                expected.\r\n"));
OldValue = 5;
NewValue = 20;
YesOrNo = AtomicCmpAndSwap(&CountIn,OldValue,NewValue); /*FALSE result*/
if(YesOrNo)
    Printf(TEXT("The AtomicCmpAndSwap function did not operate as
                expected.\r\n"));
else
    Printf(TEXT("The new value, %d, didn't replace the old value, %d,
                because the old value\r\nwasn't equal to the compared
                value, %d.\r\n"), NewValue, OldValue, CountIn);
```

See Also

AtomicSwap

AtomicDec

UINT AtomicDec(*pCount*)
UINT **pCount* // in, out

AtomicDec atomically decrements **pCount*.