

# A Windows CE Implementation of a Middleware Architecture Supporting Time-Triggered Message-Triggered Objects

Galo Gimenez  
Hewlett-Packard Company  
16399 West Bernardo Dr,  
San Diego, CA 92127, USA  
galo\_gimenez@hp.com

and

K. H. (Kane) Kim  
DREAM Lab  
UCI,  
Irvine, CA 92697, USA  
khkim@uci.edu

## Abstract

The *time-triggered message-triggered object* (TMO) programming scheme has been established to remove the limitation of conventional object programming techniques and tools in developing applications containing real-time (RT) distributed computing components. It is a unified approach for efficient design and implementation of both RT and non-RT distributed applications. As a cost-effective facility for supporting TMO-structured distributed RT programming, a middleware architecture that can be adapted to various well-established commercial software/hardware platforms has been established. It has been named the *TMO support middleware* (TMOSM). In this paper, an adaptation of TMOSM to the Windows CE platform is discussed. The internal structure of the prototype implementation, TMOSM/CE, and some implementation techniques adopted are discussed.

**Index Terms:** real time, time-triggered, message-triggered, object, middleware, TMO, TMOSM, Windows CE, operating system, distributed computing.

## 1. Introduction

The field of real-time (RT) distributed computing applications is on a rapid expansion. New challenging applications are steadily emerging. One consequence is the increasing recognition in the software technology research community of the need for an RT distributed programming technology which can be multiple times more effective in producing large-scale RT application software based on wired and wireless networks than conventional technologies are.

Quite a few on-going attempts for establishing such technologies are based in object-oriented (OO) design and implementations [IEE00, ISORC, Kim00b, OMG00, WORDS]. The *time-triggered message-triggered object* (TMO) programming scheme initiated primarily by the second co-author [Kim94, Kim97, Kim00a], is in our view one of the most promising and natural extensions of the conventional OO design and implementation techniques.

To support TMO-based design of RT computer systems in the most cost-effective manner, it is essential to establish efficient and reliable execution facilities

based on commercial-off-the-shelf (COTS) hardware and operating system (OS) platforms.

With recent advances in the OS technologies such as real-time thread constructs, some COTS OSs enable the development of middleware that supports efficient and reliable executions of application TMOs and yet is in an easily portable form. In the past four years, several prototype implementations of TMO execution support facilities based on COTS software/hardware platforms have been produced in the UCI DREAM (Distributed Real-time Ever Available Micro-computing) Lab and other cooperating research institutions. During this course, a middleware architecture that is efficient and can be adapted to various platforms has evolved. It has been named the *TMO support middleware* (TMOSM) [Kim99].

The adaptation of TMOSM to the Windows CE platform [MS00, Net01, Mur97] that is discussed in this paper, TMOSM/CE, represents our first attempt to establish TMO execution facilities based on COTS hardware + OS platforms with small footprints. An overview of the TMO structure and TMOSM will be presented in Sections 2 and 3. Then the internal structure of TMOSM/CE and some implementation techniques adopted are discussed in Section 4.

## 2. An overview of the TMO scheme and its requirements on the supporting operating system

The *time-triggered message-triggered object* (TMO) scheme was established in early 1990's [Kim94, Kim97, Kim00a] with a concrete syntactic structure and execution semantics for economical reliable design and implementation of RT systems. The TMO programming scheme is a general-style component programming scheme and supports design of all types of components including distributable hard-RT objects and distributable non-RT objects within one general structure.

TMOs are devised to contain only high-level intuitive and yet precise expressions of timing requirements. No specification of timing requirements in (indirect) terms other than *start-windows* and *completion deadlines* for program units (e.g., object methods) and *time-windows for output actions* is required. For example, priorities are attributes often attached by the OS to low-level program abstractions such as threads and they are not natural expressions of timing requirements. Therefore, no such

indirect and inaccurate styles of expressing timing requirements are associated with objects and methods [Kim94, Kim00a].

At the same time the TMO scheme is aimed for enabling a great reduction of the designer's efforts in guaranteeing timely service capabilities of distributed computing application systems. It has been formulated from the beginning with the objective of enabling *design-time guaranteeing of timely actions*. The TMO incorporates several rules for execution of its components that make the analysis of the worst-case time behavior of TMOs to be systematic and relatively easy while not reducing the programming power in any way [Kim97].

### 2.1 TMO structure and design paradigms

TMO is a natural, syntactically minor, and semantically powerful extension of the conventional object(s). As depicted in Figure 1, the basic TMO structure consists of four parts:

**ODS-sec** = object-data-store section: list of *object-data-store segments* (ODSS's);

**EAC-sec** = *environment access-capability* section: list of *gates* to remote object methods, logical communication channels, and I/O device interfaces;

**SpM-sec** = *spontaneous-method* section: list of *spontaneous methods*;

**SvM-sec** = *service-method* section.

Major features are summarized below. The second and third are the most conspicuous unique extensions of conventional object(s).

(a) *Distributed computing component:*

The TMO is a distributed computing component and thus TMOs distributed over multiple nodes may interact via remote method calls. To maximize the concurrency in execution of client methods in one node and server methods in the same node or different nodes, client methods are allowed to make non-blocking types of service requests to server methods.

(b) *Clear separation between two types of methods:*

The TMO may contain two types of methods, *time-triggered (TT-) methods* (also called the *spontaneous methods* or *SpMs*), which are clearly separated from the conventional *service methods* (*SvMs*). The SpM executions are triggered upon reaching of the real-time clock at specific values determined at the design time whereas the SvM executions are triggered by service request messages from clients. Moreover, actions to be taken at real times *which can be determined at the design time* can appear only in SpMs.

(c) *Basic concurrency constraint (BCC):*

This rule prevents potential conflicts between SpMs

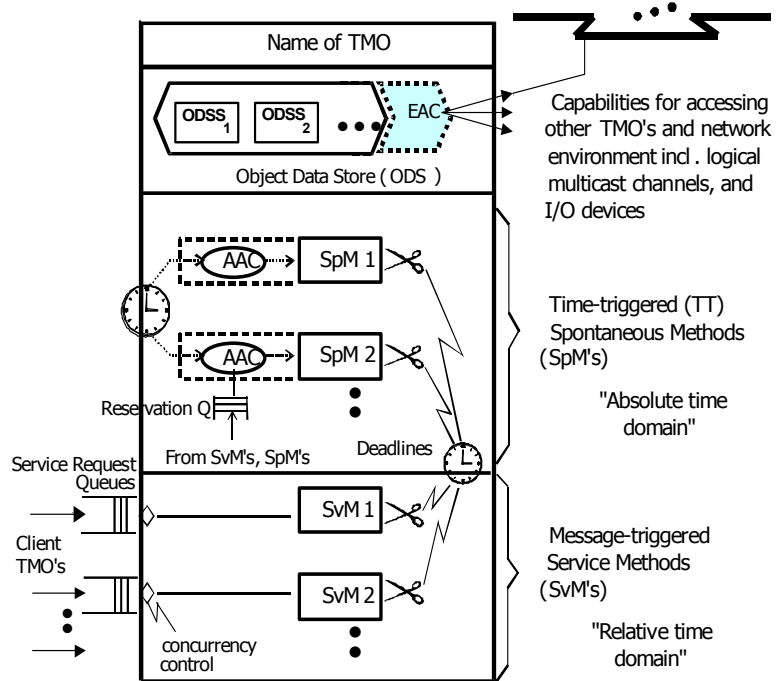


Figure 1. The basic structure of TMO (Adapted from [Kim97])

and SvMs and reduces the designer's efforts in guaranteeing timely service capabilities of TMOs. Basically, *activation of an SvM triggered by a message from an external client is allowed only when potentially conflicting SpM executions are not in place*. An SvM is allowed to execute only when an execution time-window big enough for the SvM that does not overlap with the execution time-window of any SpM which accesses the same ODSSs to be accessed by the SvM, opens up. However, the BCC does not stand in the way of either concurrent SpM executions or concurrent SvM executions.

(d) *Guaranteed completion time and deadline:*

The TMO incorporates deadlines in the most general form. Basically, for output actions and method completions of a TMO, the designer guarantees and advertises execution time-windows bounded by start times and completion times.

Triggering times for SpMs must be fully specified as constants during the design time. Those RT constants appear in the first clause of an SpM specification called the *autonomous activation condition* (AAC) section. An example of an AAC is

"for t = from 10am to 10:50am every 30min  
start-during (t, t+5min) finish-by t+10min"

which has the same effect as

{"start-during (10am, 10:05am)  
finish-by 10:10am",

"start-during (10:30am, 10:35am)  
finish-by 10:40am" }

An underlying design philosophy of the TMO scheme is that an RT computer system will always take the form of a network of TMOs. The designer of each TMO provides a guarantee of timely service capabilities of the object. The designer does so by indicating the *guaranteed execution time-window for every output* produced by each SvM as well as by each SpM executed on requests from the SvM and the *guaranteed completion time (GCT)* for the SvM in the specification of the SvM. Such specification of each SvM is advertised to the designers of potential client objects. Before determining the time-window specification, the server object designer must convince himself/herself that with the *object execution engine* (a composition of hardware, node OS, and middleware) available, the server object can be implemented to always execute the SvM such that the output action is performed within the time-window. The BCC contributes to major reduction of these burdens imposed on the designer.

## 2.2 Essential features of an operating system supporting TMOs

To support a TMOSM implementation, which in turn supports execution of application TMOs, the candidate OS must provide following capabilities:

### (a) High-resolution timer interrupts

Time-triggered actions are not only demanded by application TMOs but also used inside TMOSM for various internal operations. Time-arrival notices must be accurate and the gap between the timer's generation of a signal and the activation of a relevant computing action must be within an acceptably small bound. The OS must thus provide not only a continuously running time indicator (i.e., RT clock) but also an interval timer with sufficient precision and acceptably bounded signaling delays.

The time granularity of the RT clock, which differs among different OSs, is a major factor to be considered in selecting a platform for a specific application domain.

### (b) RT thread support

*RT thread* here is used as a generic term referring to a thread which has the following characteristics:

- (i) The delay it experiences in accessing a resource should not exceed a predetermined tight bound;
- (ii) Its execution can be interrupted only by the thread-support mechanism of the kernel (i.e., not preempted or interrupted by any other thread); and
- (iii) When a system service call is issued by an RT thread, the called system service function inherits the RT characteristics of its issuer thread.

The TMO execution engine can be implemented by expanding the OS kernel or inserting a middleware

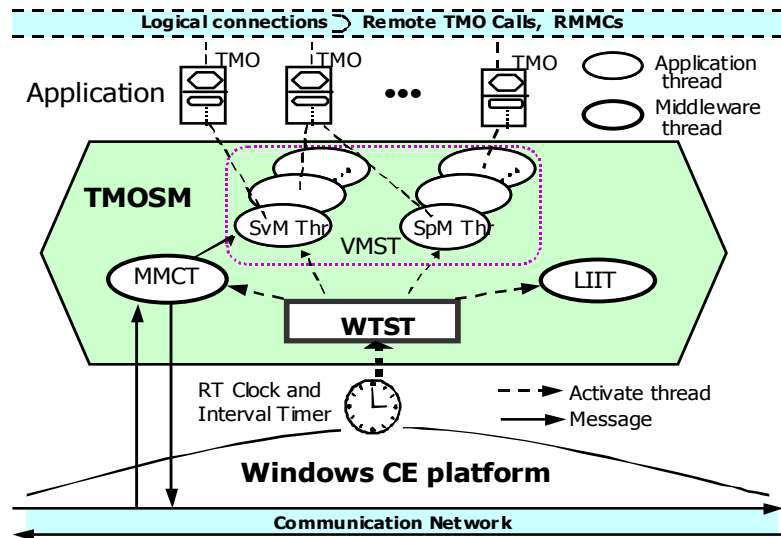


Figure 2. The basic internal thread structure of TMOSM

component layered between the OS kernel and applications. The advantages of implementing it as middleware are the following:

- (i) No change to OS kernel is needed and thus COTS kernels can be used; and
- (ii) The TMO execution engine can be easily ported onto any OS platform which has the basic features mentioned in this section.

The disadvantage of implementing it as middleware is that the middleware has to do some thread scheduling work of which the resulting decisions must be carried out by the scheduler built inside the kernel. It is thus impossible to keep the amount of execution overhead incurred to be at the same level which is incurred in the kernel expanded to support TMOs.

## 3. TMOSM: Architecture of a TMO execution support middleware

Major features of the TMOSM architecture are now briefly discussed.

### 3.1 Threads in TMOSM

Figure 2 shows the internal thread structure of TMOSM. There are three types of threads, *application threads*, *middleware (or system) threads*, and the *super-micro thread*. TMOSM assigns one application thread to each SpM or SvM of an application TMO. Middleware threads are *periodic threads* (periodically activated to run for a time-slice), each being responsible for a major part of the functions of TMOSM. In contrast to the application threads, the set and number of middleware threads is fixed at the TMOSM loading time. The authors believe that structuring of middleware threads as periodic threads is a fundamentally sound approach which leads to easier analysis of the worst-case time behavior of the object execution engine without incurring any significant

performance drawback.

The super-micro thread is called the WTST (*Watchdog Timer & Scheduler Thread*). It is a "super-thread" in that it runs at the highest possible priority level. It is also a "micro-thread" in that it manages the scheduling / activation of all other threads in TMOSM. Therefore, WTST is activated whenever a thread switching needs to be performed, e.g., upon expiration of a time-slice. Even those threads created by the node OS before TMOSM starts are executed only if WTST allocates some times-slices to them. Also, WTST checks for any deadline violations and if a violation is found, it provides an exception signal to the relevant computation unit.

The three middleware threads function as follows:

- (1) MMCT (Middleware Message Communication Thread): This periodic thread manages the sending of *middleware messages* through the communication network. Middleware messages are the messages exchanged among the middleware instantiations running on different nodes to support interaction among TMOs. MMCT also distributes middleware messages coming through the network to their destination threads.
- (2) VMST (Virtual Main System Thread): Periodically a time-slice is conceptually given to this virtual thread which merely represents all application threads running TMO methods. The actual time-slice allocations are done by WTST that executes the application scheduler function. In summary, every time-slice conceptually belonging to VMST is allocated to a fairly selected application thread.
- (3) LIIT (Local I/O Interface Thread): This middleware thread executes an I/O function pointed to by an application method. I/O functions utilize the I/O capabilities of the host node platform including serial character I/O, disk I/O, network I/O involving messages which are not middleware messages, etc. This LIIT approach makes it easier to analyze with high precision the temporal predictability of application program-segments not involving I/O.

### 3.2 Two-level scheduling

Figure 3 shows the TMOSM scheduling cycle. TMOSM adopts a two-level scheduling approach: middleware thread scheduling and application thread scheduling. WTST first executes the middleware thread scheduler to select the next middleware thread -- one among MMCT, LIIT, and VMST. From the middleware thread scheduler's point of view, all application threads in TMOSM are treated as resource-sharing children of one virtual middleware thread, VMST. When the middleware thread scheduler selects VMST as the next thread to be executed, the application thread scheduler is called to select the next application thread to use that time-slice according to the adopted application scheduling policy.

### 3.3 TTAS and AAAS

The *thread-to-thread atomic section* (TTAS) is used to avoid conflicts among middleware threads and between middleware threads and application threads when accessing shared data without using locks for the data. A thread needing to access a shared data structure orders WTST not to disturb it, i.e., orders WTST to disable the thread switch so as not to let the time-slice be taken away from the thread, until the thread notifies WTST that the disturbance prohibition period is over. So even if a thread's time-slice expires during the disturbance period (i.e. while the thread is accessing the shared data), WTST does not change the running thread. Such a code-segment is called a TTAS. The sending of a no-disturbance request to the WTST is called an "Enter-TTAS" operation, and the cancellation of a no-disturbance request is called an "Exit-TTAS" operation. The *application-to-application atomic section* (AAAS) is similar to TTAS except that the former is for atomic operation by an application thread on data shared with other application threads. So if an AAAS is in execution, WTST does not preempt current application thread to give the time-slice to another application thread, but it can give the time-slice to another middleware thread.

### 3.4 Object data store segment (ODSS)

ODSS consisting of a set of data members functions as a lockable storage unit for which an object method within the same TMO obtains a read-only or read-&-write access right. Since the access rights possessed by object methods for ODSSs determine the possibilities of concurrent execution of object methods, the segmentation of the object data into ODSSs directly impacts the degree of concurrency realizable in object executions. To facilitate dynamic decisions on the concurrent execution possibilities, each ODSS should be registered within TMOSM.

## 4. TMOSM/CE: A prototype implementation of TMOSM on Windows CE

### 4.1 Relevant features of Windows CE for supporting real time applications

The Windows CE 3.0 is an OS designed to support embedded and RT applications such as industrial controllers, personal digital assistants, smart telephones, etc. While this OS offers the developer a subset of the

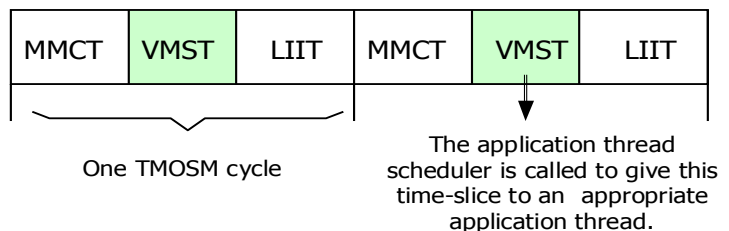


Figure 3. TMOSM scheduling cycle

common Win32 API used in Windows NT, it is based on a kernel which is different from that of Windows NT and offers RT computing support capabilities and mechanisms for space management in ROM/RAM combinations of often unusually small memory space. Windows CE also runs on a variety of processors used in embedded computing applications such as Pentium varieties, ARM, PowerPC, and MIPS processors.

Some capabilities of Windows CE that are particularly relevant to the implementation of an RT distributed computing middleware such as TMOSM include:

(1) Support of up to 256 priority levels

As a preemptive, multitasking OS, Windows CE supports up to 32 processes running simultaneously. A process consists of one or more threads. One thread in each process is designated as the primary thread. The amount of available system resources is the only factor limiting the number of threads that can be created.

The kernel's scheduler uses a priority based scheduling policy and runs threads with the same priority in a round-robin fashion. Windows CE supports a total of 256 thread priorities.

(2) Timer accuracy and thread quantum

The interval timer is a kernel object in Windows CE which encapsulates a hardware device generating an interrupt upon expiration of the preset timeout value. It can be used to enforce time-slicing scheduling. Windows CE 3.0 does not operate this timer when there are no threads to schedule. When it uses the timer, it normally sets the timer to generate an interrupt every one millisecond. Therefore if a thread calls *sleep(10)*, then the thread is expected to be awakened after  $10 + \alpha$  milliseconds where  $\alpha$  should ideally be 0 but its actual value depends on the particular machine configuration present during the sleeping period of the thread. For example,  $\alpha$  depends on the priority of the thread, the priorities of other threads, whether interrupt service routines (ISRs) are running during the sleeping period of the thread, and how long after the most recent timer interrupt the sleep call is made.

Also, in Windows CE 3.0 the thread quantum (i.e., time-slice) can be set by the application on a thread-by-thread basis, although the default quantum is 100 milliseconds and the quantum can be set to as small as one millisecond. A quantum of 0 is interpreted such that the thread will run continuously until it is blocked or preempted by a thread of higher priority.

(3) Nested interrupts and latencies

One of the most important features of the kernel that impacts the RT application performance is the ability to service an interrupt request within a specified amount of time. Windows CE splits interrupt processing into two steps: an *interrupt service routine* (ISR) and an *interrupt service thread* (IST). Each hardware interrupt request line (IRQ) is associated with one ISR. When interrupts are enabled and an interrupt occurs, the kernel calls the

registered ISR for that interrupt. The ISR, the kernel-mode portion of interrupt processing, is kept as short as possible. Its responsibility is primarily to direct the kernel to launch a relevant IST.

The ISR performs its minimal processing and returns an interrupt identifier to the kernel. The kernel examines the returned interrupt identifier and turns the associated event-signaling object, that links an ISR to an IST, on. The IST is set to wait for that event. When the kernel turns the event-signaling object on and the IST becomes the highest-priority thread ready to run, the IST starts performing its additional interrupt processing. Most of the interrupt handling actually occurs within the IST.

To prevent the loss or delay of high-priority interrupts, Windows CE 3.0 supports the nesting of interrupts based on priority if the CPU and/or additional associated hardware support it. When an ISR is running in Windows CE 3.0, the kernel disables the interrupt signal lines (hardware) which are assigned the same or lower-level priorities. If a higher-priority interrupt occurs, the kernel saves the state of the running ISR, and lets the ISR corresponding to the higher-priority interrupt run. The kernel can nest as many ISR executions as supported by the CPU.

After the highest-priority ISR ends, any pending lower-priority ISRs are executed. Then the kernel resumes processing any non-preemptible, but interruptible kernel call (Kcall) that was interrupted. If a thread was interrupted in the middle of its Kcall, execution of the Kcall resumes at this point. Once the pending Kcall is complete, the kernel lets the highest-priority ready thread start executing.

By *interrupt latency* we refer to the amount of time that elapses from the time at which an external interrupt arrives at the processor until the time at which the corresponding interrupt processing begins. Interrupt latencies in Windows CE are reasonably tightly bounded for threads locked in memory if paging does not occur. Determining the interrupt latency involves calculating the amount of time needed to invoke the corresponding ISR and IST. The *ISR latency* is the time interval from the instant when an IRQ is set at the CPU to when the ISR begins to run. The *IST latency* is the time interval from the instant when an ISR finishes execution—that is, signals a thread—to the instant when the IST begins execution.

(4) Handling of priority inversion

Priority inversion occurs when a low-priority thread owns a kernel object (semaphore, mutex, etc) that a higher-priority thread requires. Windows CE deals with priority inversion using *priority inheritance*, where a thread that is blocked holding a kernel object needed by a higher-priority thread inherits the higher priority. Priority inheritance enables the lower-priority thread to finish the use of and free the locked resource for subsequent use by the higher-priority thread. In previous versions, the kernel was designed to handle an entire inversion chain. In Windows CE 3.0, the kernel is designed to handle

priority inversion to the depth of only one level at all times.

## 4.2 Implementation of TMOSM on Windows CE

### 4.2.1 Implementation of a high-resolution timer for TMOSM

WTST is the TMOSM's thread scheduler, which is triggered by one of the following events:

- (i) "exit\_TTAS";
- (ii) "exit\_AAAS";
- (iii) Completion of a TMO method execution;
- (iv) Entry of a TMO method execution into a blocked state; and
- (v) a timeout.

When one of these events is signaled, WTST will be awakened to schedule the next thread. If the event signaled is "TMO method completion" or "TMO method blocked", WTST simply gives the rest of the time-slice to the first application thread in the ready queue as a bonus. If the signal is "exit\_TTAS" or "exit\_AAAS", WTST will check whether the current thread has used up its time-slice share; if so, it will schedule the next thread, and otherwise, it will let the current thread resume.

To avoid preemption of TMOSM by other threads unrelated to TMOs and to satisfy the demanding timing requirements of TMOs, it is necessary to implement threads of TMOSM to run at the priority level that is higher than the priorities of other threads. Also, WTST enslaves the Windows CE's built-in scheduler and works as the scheduler of TMO applications. Every time WTST selects one middleware or application thread, it will suspend all other threads and itself. The priority of WTST is the highest and all other middleware and application threads in TMOSM are the second highest to guarantee that they will not be preempted by other Windows CE threads. When the selected thread's time-slice expires, WTST will be awakened to select another thread.

TMOSM model requires WTST to be activated periodically by a high-precision timer. This is to construct a time-slicing mechanism. Windows CE encapsulates a hardware timer into several types of timer objects. However, it lacks a timer object that can be set to periodically enter the "signaled state", an event for which a thread can wait. In contrast, Windows NT provides such a timer object. So, in our first prototype implementation we used the timeout parameter associated with the API *WaitForMultipleObjects* to build a time-slicing mechanism.

Suppose WTST properly sets a timeout by calling *WaitForMultipleObjects* with a correct parameter. Then WTST enters a mode in which it waits for a timeout event. The timeout event will occur at the end of the current time-slice and will awaken WTST. Suppose the adopted size of a time-slice is 6 milliseconds and the last time-slice started at 10AM. The current time-slice was

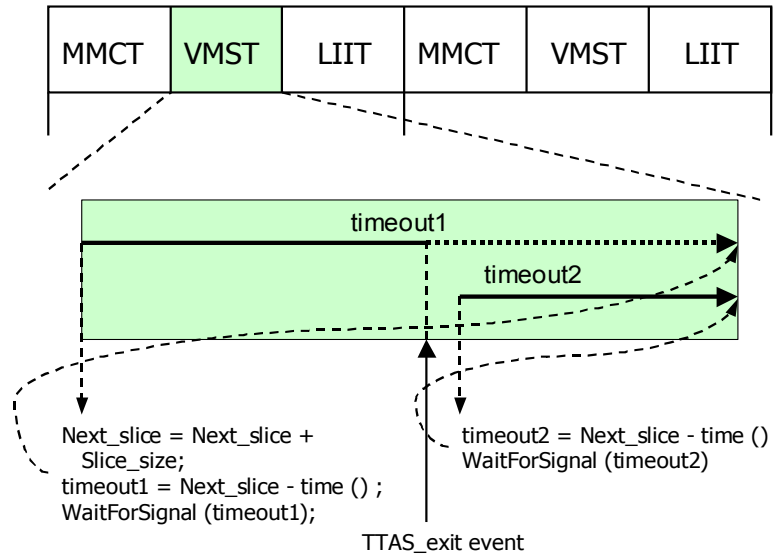


Figure 4. Timeout mechanism

supposed to start at 6 milliseconds past 10 AM and it must end at 12 milliseconds past 10AM. The awakened WTST will perform its normal duties expected at the beginning of each time-slice. Thereafter, WTST will need to call *WaitForMultipleObjects* with a correct parameter value and thus it needs to calculate how much time remains until 12 milliseconds past 10AM. To do this WTST checks the current real time maintained by the high-performance counter in Windows CE which encapsulates another hardware component. The high performance counter in Windows CE provides the current time to the precision that is at least one of a millisecond-level but depends on the hardware platform.

If WTST finds the current real time to be 7 milliseconds past 10 AM, then WTST calls *WaitForMultipleObjects* with the timeout parameter set as 5 milliseconds. Or if the current real time is 8 milliseconds past 10AM, WTST uses the timeout parameter set as 4 milliseconds.

On the other hand, the timeout is not the only event for which WTST may be waiting. For example, if a TMO method execution exits from a TTAS under certain circumstances, a signal will be generated to awaken WTST. This will nullify the timeout that WTST established through a *WaitForMultipleObjects* call. The awakened WTST must then take a scheduling action (related to the state changes of the middleware threads) and also re-establish the timeout via a new *WaitForMultipleObjects* call.

Consider again the case where the last time-slice was supposed to start at 10AM. If WTST finds the current real time to be 4 milliseconds past 10 AM and its decision is to let the current middleware thread continue, then WTST calls *WaitForMultipleObjects* with the timeout parameter set as 2 milliseconds. On the other hand, if the scheduling decision is to donate the remaining 2

milliseconds to the next thread, which is normally entitled to a time-slice of 6 milliseconds, then WTST calls *WaitForMultipleObjects* with the timeout parameter set as 8 milliseconds. Similarly, if the last time-slice was supposed to start at 6 milliseconds past 10AM, the current real time is 7 milliseconds past 10AM, WTST uses the timeout parameter set as 5 milliseconds to let the current thread continue to use the remaining share of its time-slice.

Therefore, WTST can calculate the new timeout value that should be used as a parameter in the next *WaitForMultipleObjects* call, using the code that appears below:

```
timeout = DCSAge_slice_start +
         slice_size -
         ClockService.GetCurrentDCSage();
```

Figure 4 depicts this situation. Therefore under this approach the TMOSM/CE implementation keeps track of the real time at which the most recent time-slice began.

#### 4.2.2 The ClockService class

The Windows CE implementation of TMOSM contains a module, ClockService class, which provides a way to access the high-performance counter provided by the system. The API *QueryPerformanceCounter* can be used in accessing an OEM-supplied high-performance counter. The resolution of this counter can be retrieved by calling the *QueryPerformanceFrequency* function. The resolution value will depend on the hardware implemented by the OEM. The default implementation of Windows CE that comes in most current commercial devices gives a resolution of 1 millisecond.

#### 4.2.3 Unicode

Windows CE uses Unicode in its APIs. While TMOSM/NT uses the MCBS encoding, TMOSM/CE uses this Unicode encoding in all of its code. To facilitate interoperability between the TMOs running on TMOSM/CE and the TMOs running on TMOSM/NT or other TMOSM platforms based on MCBS or single-byte encoding, the communications into and out from TMOSM/CE nodes are done using the MCBS encoding. To accomplish that, Unicode-to-MCBS translations are done in all outgoing messages and the opposite translations are done in all incoming messages.

#### 4.2.4 Program initiation

TMOSM/CE consists of two dynamic linked libraries (DLLs): TMOSM.dll and TMOSL.dll. TMOSM.dll provides main

service functions and TMOSL.dll (TMO support library) is a friendly application programming interface (API) which wraps around the services of TMOSM and provides easy-to-use building-blocks for TMO programmers. In addition, TMOSL.dll contains one special function incorporated for use in this Windows CE adaptation, WinMain() function. When an application is called for in an Windows CE platform, the OS looks for the WinMain function as soon as the application code is loaded into the memory. WinMain is normally designed to take the following actions:

- (1) Common initialization action which includes the creation and registration of the main application window;
- (2) Spawning of a new high-priority thread and letting the thread to start executing the application structured as a TMO network; and
- (3) Initiation of an event-signal polling loop which is a permanent loop designed to check the special queue, which may be called the event-signal queue, and process any event-signal found there.

Event-signals come generally from the CE OS and their examples are user-interface redrawing order, application shutdown order, etc. The step (2) above involves calling the main function of a TMO-structured application, i.e., TMOMain(). Since TMOMain is used as the standard entry function name, the code for step (2) above is application-independent. In fact, none of the three steps above need to be tailored to any TMO-structured application. Therefore, it is natural to include the WinMain function into TMOSL.

Once an application structured as a network of TMOs is constructed, say, TMO\_App.exe, it is linked together with TMOSL.dll and becomes a single load module.

Figure 5 depicts various steps involved in

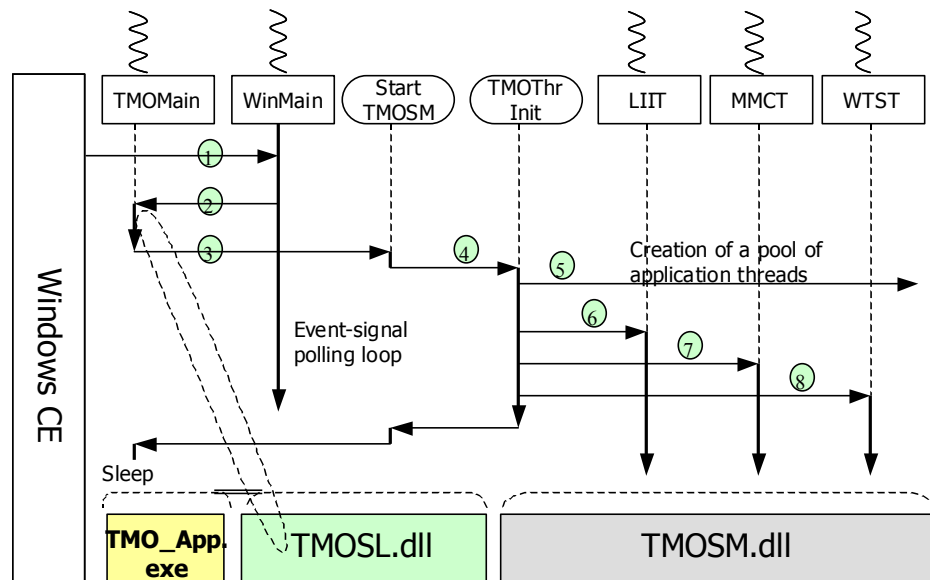


Figure 5. TMOSM/CE initialization

initialization of TMOSM and application TMOs in some detail.

Step 1: This occurs right after a new application is called either via a manual input of a command or as a startup application. This step involves loading the package consisting of TMO\_App.exe and TMOSL.dll into the working memory, creating a new thread, and letting the thread start executing WinMain(). WinMain first takes the common initialization action discussed earlier in this section.

Step 2: WinMain creates a new high-priority thread and lets thread start executing TMOMain(). In its beginning part, TMOMain typically contains statements for instantiation of TMOs. As such statements are executed, TMOSM.dll is called in. The instantiated TMOs are registered into TMOSM but they are not executed yet since execution of the TMOs requires actions of WTST and middleware threads of TMOSM and those threads have not been created yet. Meanwhile, WinMain run by a dedicated thread progresses to enter an event-polling loop which was discussed earlier in this section. The loop is a permanent loop and the thread in this loop will go forward whenever WTST releases a left-over time-slice (i.e., the part or entirety of a time-slice not needed by any middleware thread) to the threads not related to TMO executions.

Steps 3 - 8: TMOMain calls the function StartTMOSM() in TMOSL.dll and the latter function in turn calls the TMO thread initialization (TMOThrInit) function in TMOSM.dll. The TMOThrInit function then creates a pool of application threads, two middleware threads (MMCT and LIIT), and finally WTST (without activation). Thereafter, TMOThrInit run by the thread which started TMOMain activates WMST and returns the thread control to StartTMOSM which in turn returns the thread control to TMOMain. TMOMain immediately enters the sleeping mode.

TMOSL also provides helper functions for debugging applications and logging events. When working with a debugging version of the code, log messages appear on the desktop debugging console, facilitating the development of applications in devices that lack proper displays. This is implemented using the Embedded Visual C++ remote debugging functionality.

### 4.3 Implementation Results

The TMOSM/CE middleware was developed using the Embedded Visual C++ 3.0 toolset and the PocketPC software development kit (SDK), both provided by Microsoft. During the construction of the prototype, the emulation environment of a PocketPC was used. This emulation environment provided as a part of the PocketPC SDK, implements a Windows CE application device emulator running on the Windows 2000 OS.

In this environment we executed some sample application TMOs. For example a simple test application that runs two TMOs, each with an SpM and an SvM, is

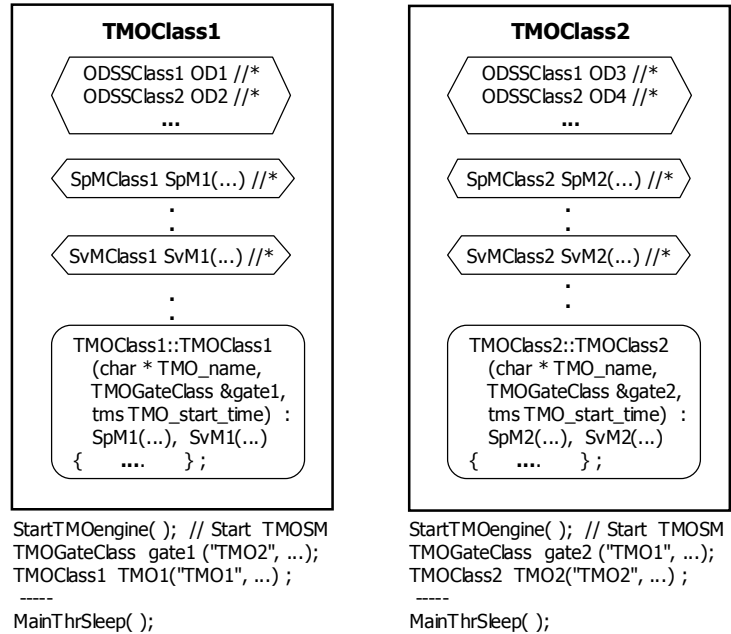


Figure 6. Sample TMO application

shown in Figure 6. The two TMOs running in two different nodes exchange messages periodically. Each SpM in one TMO calls an SvM in the other TMO.

This emulation-based execution environment is limited in facilitating measurements and studies on timing performance of application TMOs. To mention some of those:

(1) The emulation environment imposes a performance penalty on the application. Also, NT processes other than the CE device emulator compete with the latter for using machine resources. In addition, ISRs in NT device drivers interfere with the execution of the CE device emulator. These undesirable interferences would be absent or minimally present on actual devices because an OEM is not likely to put such interfering items into actual CE devices. However, there is a process which actively uses execution resources and cannot be turned off. That is the basic CE system process composed of threads running the memory manager, cache manager, etc. A good understanding of the CE architecture is important in realizing full guarantee on the timeliness of TMOSM/CE.

(2) We also used the mechanism provided by the debugging tool of the Embedded Visual C++ development environment to redirect logging statements to a file in the hosting computer. This involves continuous communication between the emulator and the hosting computer, which has a great impact on performance. This mechanism is of course disabled on the "release" version of the middleware

Even with all those caveats, the overall performance results were quite positive. The basic-time slice for a middleware thread is 6 milliseconds. VMST receives a basic time-slice every 18 milliseconds and gives it to a

selected application thread (executing a TMO method). VMST switches to a new application thread after giving four time-slices to one application thread. This test environment based on a simple 300Mhz Pentium computer has not been much optimized yet. We have observed that it can accurately follow the specification of a time-window for activating a method to the 80-millisecond level of precision.

The first version of the TMOSM/CE middleware prototype has the total code size of 176 Kbytes. The test application with two simple TMOs had a total footprint in memory of 5 Mbytes; this includes the environment running a standard Windows CE device emulator, the middleware, and the application itself. Upon initialization, the middleware makes reservation of enough resources (threads, queues) to run complex applications developed in our lab. Tuning on the amount of resources reserved can be done for specific applications but this aspect needs to be studied further. In particular, it should be possible to reduce the memory footprint of the middleware substantially, if not drastically. A series of experiments along this line are in plan.

## 5. Summary

An adaptation of the TMOSM middleware architecture to the Windows CE platform, TMOSM/CE, has been discussed. In this paper we presented the internal structure of and some implementation techniques used in the prototype TMOSM/CE that has been tested with a Windows CE device emulator only. However, the adaptation of the prototype TMOSM/CE to iPAQ devices manufactured by Compaq Corporation and equipped with 802.11 wireless LAN connections is about to start and will be carried out in the rest of 2001. It will then be possible to study in greater depth the performance improvements in such environments due to the elimination of interferences from the logging mechanism and other unique parts of the emulation environment. In addition, experimental efforts will continue toward making assessments of the ease of RT distributed programming enabled by TMOSM/CE.

**Acknowledgment:** The research reported here was supported in part by the NSF under Grant Numbers 99-75053 (NGS) and 00-86147 (ITR), and in part by the US DARPA under Contract F33615-01-C-1902 monitored by AFRL. No part of this paper represents the views and opinions of any of the sponsors mentioned above.

## Reference

[IEE00] 'A special issue of *Computer* (a magazine of IEEE Computer Society) on Object-oriented Real-time distributed Computing', June 2000.

[ISORC] *ISORC (IEEE CS Int'l Symp. on Object-oriented Real-time distributed Computing)* Series; 1st held in April 1998, Kyoto, Japan; 2nd in May 1999, St. Malo, France; 3rd in March 2000, Newport Beach, CA; 4th in May 2001, Magdeburg, Germany. Proceedings are available from IEEE CS Press.

[Kim94] Kim, K.H. et al., "Distinguishing Features and Potential Roles of the RTO.k Object Model", *Proc. 1994 IEEE CS Workshop on Object-oriented Real-time Dependable Systems (WORDS)*, Oct. '94, Dana Point, pp.36-45.

[Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, Vol. 30, No.8, August 1997, pp. 62-70.

[Kim99] Kim, K.H. Ishida, Masaki, Liu, Juqiang, "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation", *Proc. ISORC '99 (2nd IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing)*, St. Malo, France, May, 1999, pp.54-63.

[Kim00a] Kim, K.H., "APIs Enabling High-Level Real-Time Distributed Object Programming", *IEEE Computer*, June 2000, pp.72-80.

[Kim00b] Kim, K.H., "Object-Oriented Real-Time Distributed Programming and Support Middleware", *Proc. ICPADS 2000 (7th Int'l Conf. on Parallel & Distributed Systems)*, Iwate, Japan, July 2000, pub. by IEEE CS Press (Keynote paper), pp.10-20.

[MS00] '*Designing and Optimizing Microsoft Windows CE 3.0 for Real-Time Performance*', Microsoft Corporation, Sept. 2000.

[Mur97] John Murray. "Real-Time Systems with Microsoft Windows CE". Microsoft Corporation, Sept. 1997.

[Net01] Netter, C.M., and Bacellar, L.F., "Assessing the Real-Time Properties of Windows CE 3.0", *Proc. ISORC 2001 (4th IEEE CS Int'l Symp. on Object-Oriented Real-time Distributed Computing)*, Magdeburg, Germany, May, 2001, pp. 179-184.

[OMG00] 'Collection of Slide Presentations, 1<sup>st</sup> OMG Workshop on Real-Time / Embedded Distributed Object Computing', July 2000, Crystal City, VA.

[WORDS] *WORDS (IEEE CS's Workshop on Object-oriented Real-time Dependable Systems)* Series; 1st held in Oct. '94, Dana Point; 2nd in Feb. 1996, Laguna Beach; 3rd in Feb. 1997, Newport Beach; 4th in Jan. 1999, Santa Barbara; 5th in Nov. 1999, Monterey; 6th in Jan. 2001, Rome, Italy. Proceedings are available from IEEE CS Press.