

## Commanding and Reactive Control of Peripherals in the TMO Programming Scheme

K. H. (Kane) Kim

University of California  
Irvine, CA

[khkim@uci.edu](mailto:khkim@uci.edu), <http://dream.eng.uci.edu/>

**Abstract:** Although high-level real-time distributed computing objects are generally written in forms independent of execution platforms, input and output (I/O) activities involving peripherals are inherently platform-dependent. Yet, writing parts of real-time objects for controlling peripherals should be done in forms compatible with the adopted real-time object programming styles. Basic issues are discussed in the context of an object-oriented real-time programming scheme called the *time-triggered message-triggered object* (TMO) programming scheme. A desirable goal here is to facilitate both commanding and reactive control of peripherals in TMOs in general forms while enabling relatively easy analysis of the timing behavior of such TMOs. This paper presents several techniques to meet these requirements.

**Keywords:** real time, embedded, TMO, time triggered, message, object, middleware, distributed, parallel, programming.

### 1. Introduction

High-level approaches to real-time (RT) distributed programming are generally aimed for relieving the programmer from the burden of handling many details specific to execution platforms which may be composed of processors, standard peripherals, operating system (OS) kernels, and middleware [Bol00, IEE00, Kim97, Kim00b, OMG00a, OMG00b, Sch00, Sel00]. The time-triggered message-triggered object (TMO) programming scheme is among the most advanced approaches of such kind [Kim97, Kim00a]. The support tools for the TMO scheme can be based on well-established OO programming languages such as C++, C#, and Java and on ubiquitous commercial real-time OS kernels or even on the Microsoft Windows family of OS kernels. Being a high-level programming approach, the TMO scheme offers the following benefits to complex distributed application designers:

- (TB1) The TMO model is a natural and syntactically small extension of the conventional object model(s) such that typical object-oriented (OO) programmers can adopt it with relatively small efforts.
- (TB2) The TMO scheme enables RT programming and distributed programming in a highly abstract and yet high-precision form, relieving the programmer of the burden of dealing with underlying OS services

and network protocols and allowing the programmer to be highly productive and focused on essential design activities.

- (TB3) The scheme enables systematic design-time guaranteeing of timely service capabilities of objects.
- (TB4) The scheme facilitates uniform structuring of (i) both RT and non-RT distributed application systems, (ii) both control computer systems and their application environment simulators [Kim97], and (iii) requirement specifications and system designs arising at various phases of system engineering.

Feasibility studies have been conducted over several years and the results obtained strongly indicate the potential of the TMO scheme in meeting its goals in the practicing field.

Currently one of the most advanced models of the execution engines for TMOs is the middleware architecture named the *TMO support middleware* (TMOSM) which can be implemented on widely used commercial hardware and OS platforms [Kim99, Kim01]. TMOSM uses well established services of commercial OSs, e.g., process and thread support services, short-term scheduling services, and low-level communication protocols, in a manner transparent to the application TMO programmer. Prototype implementations based on several most popular OS kernels such as Windows NT/XP, Windows CE, and Linux exist.

A friendly programmer interface wrapping the services of TMOSM has also been developed and named the *TMO Support Library (TMOSL)* [Kim00a]. It consists of a number of C++ classes and is meant to be an API-style approximation of an idealistic fully featured TMO programming language. It empowers C++ programmers with the following powerful and natural mechanisms for specification of unique and essential features of any RT distributed programs:

- (1) Global time base;
- (2) Time-triggered (TT) action;
- (3) Concurrency control;
- (4) Interaction among RT objects and RT message communication, including blocking calls for remote methods, non-blocking calls, and direct message communication over *programmable logical multicast channels*.

Although the TMO scheme is a high-level RT object, input and output (I/O) activities involving peripherals are

inherently platform-dependent. Also, as computer-embedded communication-equipped devices proliferate, intelligent sensors and other peripherals are also appearing at an increasing rate. Such peripherals increasingly operate in autonomous modes and require application-specific prompt response to occurrences of certain conditions in their states. This means that some parts of the application TMOs running on the core (CPU and memory) of the execution platform should be designed to do *reactive control*, i.e., react to special condition occurrences in such peripherals rather than controlling them by continuously issuing detailed commands at a high rate. Therefore, TMOSM must have general facility for information flow from application TMOs to the components of commercial OS kernels encapsulating peripheral devices and information flow in the opposite direction.

A desirable goal here is to facilitate both commanding and reactive control of peripherals in TMOs in general forms while enabling relatively easy analysis of the timing behavior of such TMOs. Several basic styles of writing TMO parts for controlling I/O activities have been devised to meet those requirements and are presented in this paper. The TMOSM architecture has also been extended in recent years to incorporate a group of threads for use in executing I/O jobs. A newly developed programming technique which enables designing interactions between those threads dedicated to I/O jobs and the threads executing non-I/O parts of TMO methods, is also presented.

In the next section, an overview of the TMO programming scheme and the latest version of the TMOSM architecture is given. Then in Section 3, basic styles of designing commands to peripherals in the TMO programming environment centered around TMOSM and TMOSL are discussed. The basic styles of designing reactive controls of peripherals are discussed in Section 4. The paper concludes in Section 5.

## 2. An overview of the TMO scheme and the TMOSM architecture

The *time-triggered message-triggered object* (TMO) scheme was established in early 1990's [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.

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.

### 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 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 for method execution and deadline for result return:*

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. In addition, deadlines can be specified in the client's calls for service methods for the return of the service results.

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, which may be produced in a top-down multi-step fashion [Kim97]. 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.

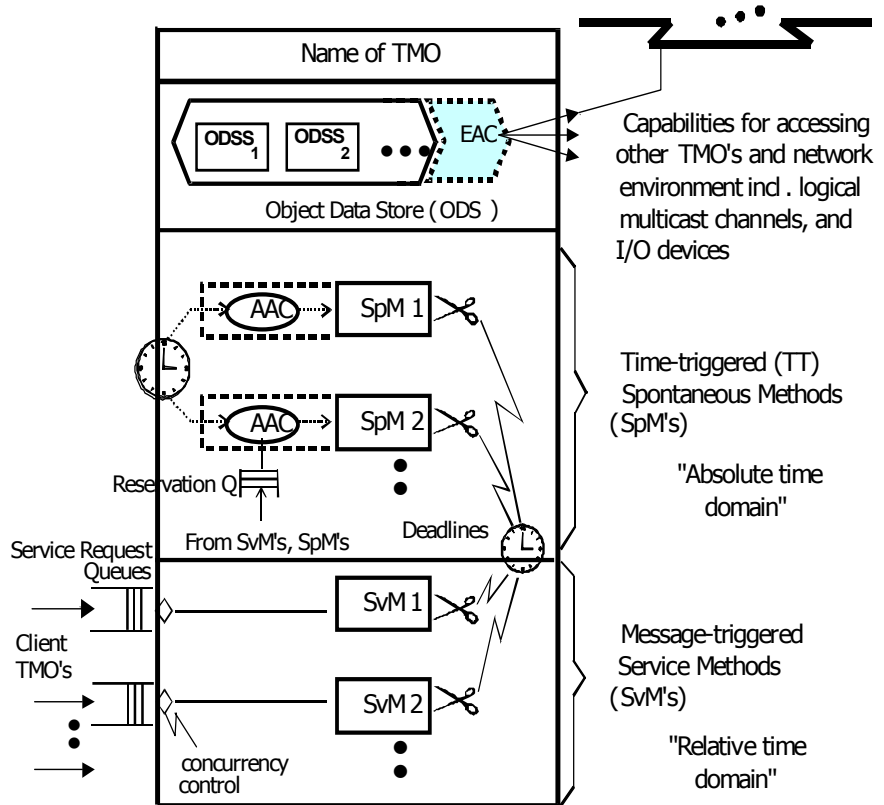


Figure 1. The basic structure of TMO (Adapted from [Kim97])

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 TMO support middleware (TMOSM)

A cost-effective way to support execution of TMOs is to realize an execution engine by developing a middleware running on well established commercial software / hardware platforms [Kim99, Kim01, Sho98]. The most obvious and important requirement that a TMO execution engine must meet is to accurately honor the timing specifications associated with various application program-segments, in particular, TMO methods. Although major structural characteristics of TMOSM have not been changed since its inception in 1998, its refinements have steadily occurred, especially in concurrency and I/O management.

### 2.2.1 Internal thread structure

The internal thread structure of TMOSM is shown in Figure 2. This architecture has been devised to enable relatively easy analysis of the worst-case execution times of TMO methods.

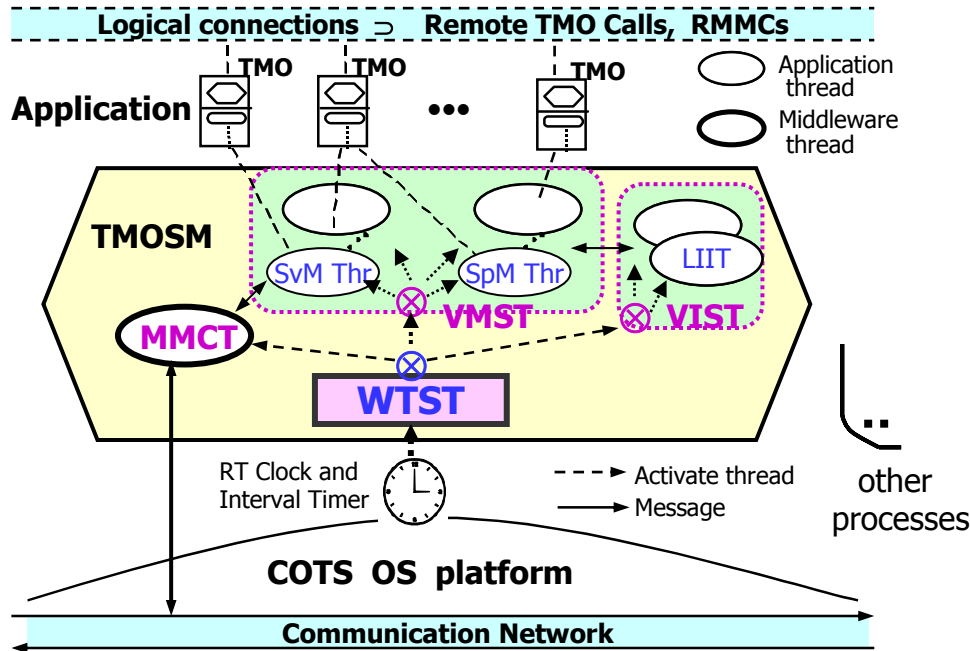


Figure 2. The basic internal thread structure of TMOSM

TMOSM consists of three types of threads, *application threads*, *middleware 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. 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. Therefore, WTST is activated whenever thread switching needs to be performed, e.g., upon expiration of a time-slice. 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) **VIST** (*Virtual I/O System Thread*): This virtual thread maintains a pool of threads which are called *local I/O threads* (LIITs) and execute the I/O requests from application threads. The time-slices allocated to VIST are actually distributed to LIITs. Each LIIT is assigned to execute an I/O function that utilizes 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 VIST approach has been motivated by the desire to make it easier to analyze with high precision the temporal predictability of application program-segments not involving I/O (and, to a less extent, the temporal predictability of I/O activities).

(3) **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. Therefore, every time-slice conceptually belonging to the VMST is allocated to a fairly selected application thread.

Several features of this TMOSM architecture contribute to simplifying the analysis of the execution time behavior of application TMOs running on TMOSM. First, the strictly periodic nature of middleware threads and the dedication of each middleware thread to a specific functionality enable largely independent analysis of the

part of the execution time behavior of application TMOs that depends on a particular middleware thread. For example, in computing the maximum time taken for transmitting a message  $\alpha$  in a queue attached to MMCT to a queue attached to the MMCT in a remote node, one needs to focus on a few factors only: the number and sizes of the messages in the queue ahead of  $\alpha$ , the size of  $\alpha$ , guaranteed bandwidth of the network path between the source and the destination, and the frequency and the size of the time-slice given to MMCT in each node.

Secondly, the execution time of an I/O can be in general specified, analyzed, and measured in a larger time gain than that which can be used in specifying and analyzing the computation relying on the CPU only. Therefore, dedicating LIITs to handling such I/O activities enables the high-precision analysis of the execution time behavior of the CPU-intensive computation.

### 2.2.2 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, VIST, 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. VIST behaves similarly.

VMST (or rather the application thread scheduler) may choose an application thread running a TMO method consecutively for several TMOSM cycles. That is, VMST may let an application thread use several (e.g., four) time-slices in a row before it lets another application thread use a time-slice. In a sense, the group of time-slices given consecutively to a TMO method execution thread may be viewed as forming an application-thread-level time-slice. Similarly, VIST may let an LIIT use several time-slices in a row before it lets another LIIT use a time-slice coming to itself (i.e., VIST). In general, it is sensible to make the group of time-slices given consecutively to an LIIT to be larger than the group of time-slices given to a TMO method execution thread.

Other architectural aspects of TMOSM which are useful in realizing the highly predictable behavior of the middleware are not discussed here due to space limit [Kim99]. From the validation tests conducted, we have found that our prototype implementation of TMOSM on the Windows XP platform, TMOSM/XP, can accurately enact the time-window for activating a method as small as

10ms and the method completion deadline as short as 3ms unless a device driver not preemptible for an excessively long time gets involved.

## 3. Basic styles of writing commands to peripherals into TMOs

Since the APIs available to TMO programmers consist of the part wrapping the services of not only the middleware, i.e., TMOSM, but also a commercial OS kernel, commands to peripherals can be written into any TMO method as simple calls for relevant APIs. One style that most typical C++ programmers are not practicing yet but is strongly suggested for them is to declare wrappers of the APIs for commanding and controlling peripherals in the special section of the ODS called the EAC (*environment access capability*) section (i.e., as a special data member). This is depicted in Figure 4.

The wrappers are essentially objects in which methods are included for invoking the kernel APIs for passing commands to peripherals. This style would enable TMO program readers to easily recognize which

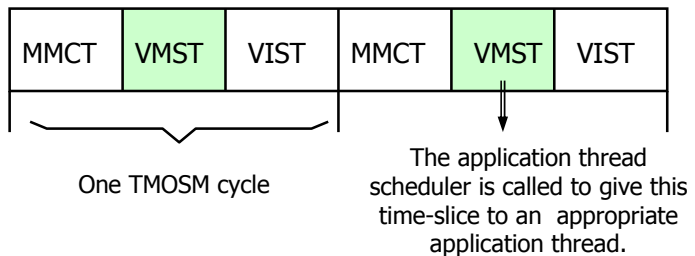


Figure 3. TMOSM scheduling cycle

non-standard peripherals are used. Moreover, if such wrappers are declared as ODSSs, then harmonious sharing of wrappers by concurrently running TMO method executions is almost automatically achieved since ODSSs to be used during a TMO method execution are generally locked at the time of initiating the method execution.

## 4. Basic styles of writing reactive control of peripherals into TMOs

In this section, four different basic styles of writing reactive control of peripherals into TMOs are discussed.

### RC1: Polling by a dedicated SpM

In a strict sense, this is not an approach for reactive control. The approach is to dedicate an SpM for monitoring the state of the peripheral and record on occurrence of a special condition into an ODSS (ODS segment which is an atomically accessible group of data members). If main application functionalities are embedded into other SpMs and SvMs, then one can view

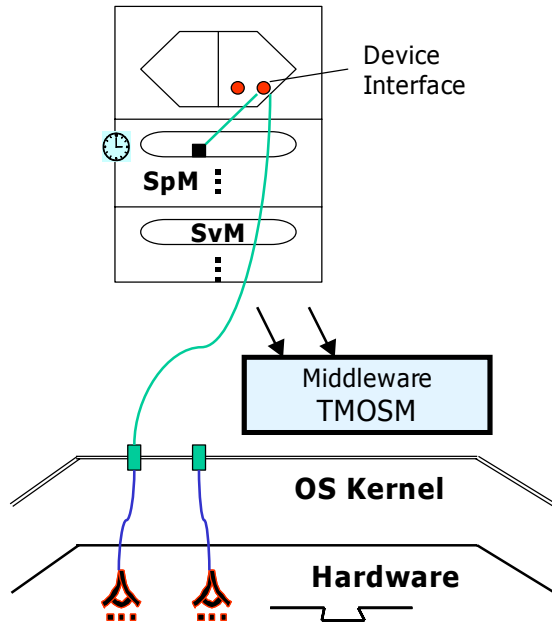


Figure 4. Peripheral commands from a TMO via a device interface in the ODS

the SpM dedicated for polling and monitoring the peripheral as a slave function, thereby qualifying this approach as one for reactive control. That is, the information (or signal) flow from the peripheral to main application functionalities is asynchronous.

The dedicated SpM checks the state of a peripheral through I/O APIs provided by the (commercial) kernel part of the TMO execution engine. Although this is conceptually the simplest approach and can be good enough in a large number of applications, the amount of execution engine resources consumed by the dedicated SpM can become a factor of severe concern in certain demanding RT applications.

### RC2: Polling by a local I/O interface thread (LIIT) and interactions between an MET and an LIIT

The nature of LIITs was discussed in Section 2. Each LIIT is assigned to execute an I/O function that involves calls for I/O APIs provided by the OS kernel. TMO methods are executed by dedicated threads called *method execution threads* (METs). When a TMO method calls the following library function, TMOSM assigns the I/O function written by the TMO programmer and specified as a parameter in this function call is assigned to an LIIT for execution.

```
Use_LIIT ( IO_func_addr, *param, size(param),
time_stamp, fault_conclusion_time ),
```

where "IO\_func\_addr" is the name or address of the I/O function that is written by the TMO programmer and involves use of I/O APIs provided by the OS

kernel, "\*param" and "size(param)" describe the data structure (containing the parameter-set) to be used by the LIIT as the actual parameter in calling IO\_func,

"time\_stamp" keeps the time stamp created at the time of executing this Use\_LIIT call, and "fault\_conclusion\_time" specifies the time-out value for the event of the execution result of the IO\_func becoming available for pickup by the TMO method which makes this Use\_LIIT call.

The TMO method which made the Use-LIIT call can check the completion status of the LIIT's execution by use of the following library functions.

```
Blocking_check_LIIT_completion_status (
time_stamp, ... )
```

```
Non_blocking_check_LIIT_completion_status (
time_stamp, ... )
```

The parameter-set passed onto the LIIT may include an ODSS that can hold the data from the peripheral. Note that a TMO method may issue multiple Use\_LIIT calls. A TMO method can certainly make several LIITs execute different IO functions, all in parallel.

The approach of using LIITs has been motivated by the desire to make it easier to analyze with high precision the temporal predictability of application program-segments not involving I/O (and, to a less extent, the temporal predictability of I/O activities).

The I/O function executed by the LIIT can be designed to perform polling of the peripheral. Upon detection of a special condition in the peripheral, the I/O function can make a record in a specified ODSS. Again, since TMO methods carry main application functionalities, one can view this as an approach for implementing reactive control of peripherals. Compared to the previous style RC1, this style of relying on an LIIT can lead to better performance in execution of TMO methods in many cases since METs and LIITs never compete for the same time-slice. A more important benefit is in the ease of the analysis of the execution times of TMO methods in general.

An LIIT commissioned by an MET may interact with the MET before it completes its job and terminates. Such interactions are essentially asynchronous in nature. Figure 5 depicts a typical arrangement for facilitating such interactions. A TMO method owns a circular buffer with an update counter. The TMO method has read-write access rights for the circular buffer and the update counter and the I/O function has read-only access to those. Similarly, the I/O function owns a different circular buffer with an update counter.

Access to each update counter is controlled under the *non-blocking writer* (NBW) scheme discussed in [Kop97]. For example, consider the update counter and the circular buffer owned by the TMO method. When the

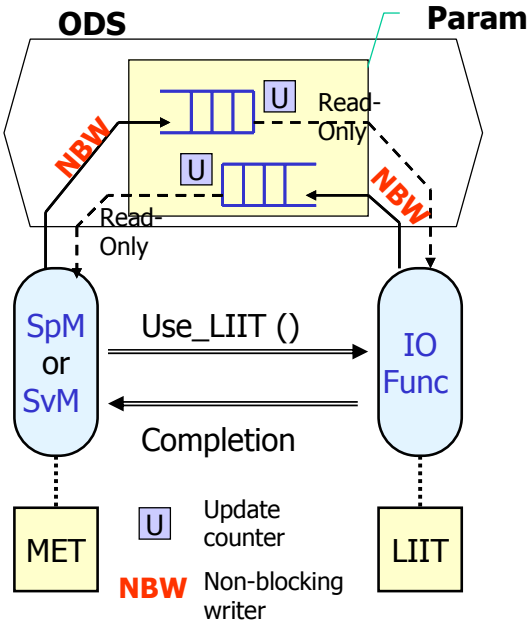


Figure 5. Interactions between MET and LIIT

TMO method inserts a data item into the circular buffer, it first increments the associated update counter and then inserts the data item. The I/O function cannot interfere with the TMO method during this writing.

When the I/O function accesses the circular buffer owned by the TMO method, the I/O function first reads the update counter of the TMO method under the NBW scheme. The amount of time taken by the I/O function in successfully reading the update counter varies depending on whether the TMO method interferes, i.e., writes into its update counter, or not while the I/O function tries to read it.

Once the I/O function successfully reads the update counter of the TMO method, the I/O function compares the newly read value of the update counter against the value of the update counter that it saw last time. Through this comparison, the I/O function knows whether there is any new data item which it did not see before. If the update counter indicates that there are some new data items in the buffer, the I/O function reads those data items without any interference from the owner, i.e., the TMO method, because of the following. First of all, the buffer is a circular buffer. Secondly, if the TMO method needs to access the buffer, it will be accessing the buffer slots different from those being accessed by the I/O function because of the protocol that is used between the TMO method and the I/O function and is similar to the sliding-window protocol.

To be more specific, after the I/O function

successfully reads items in the circular buffer owned by the TMO method along with the content of the associated update counter, it inserts into its own circular buffer a copy of the content of the update counter of the TMO method. Note that the copy gives a clear hint on what recent data item(s) in the buffer of the TMO method was read by the I/O function. Before the TMO method inserts new data items into its buffer, it reads the update counter and the buffer of the I/O function. From the data items read from the buffer of the I/O function, the TMO method learns which slots in its own buffer were read last time by the I/O function. To the data items to be inserted into its own buffer, the TMO method adds a copy of the content of the update counter of the I/O function. If the TMO method finds out that its own circular buffer is full and none of the buffer slots can be erased, then the TMO method will wait for some time and check the update counter and the buffer of the I/O function again.

### RC3: TMO-pretender interface to peripherals

This approach is applicable to the peripheral with programmable interface hardware. In this approach, a program is written into the interface hardware and the program imitates a simplistic TMO. Therefore, such a programmed interface is called a *TMO pretender*. A TMO pretender is a function implemented such that it can generate a message carried over the same channel used for inter-TMO communication and accepted by a real TMO (i.e., SvM). The responding SvM can then record some information into an ODSS or provide a response to the TMO-pretender peripheral interface.

In Figure 6, each sensor has a programmable interface and the interface is programmed to function as if it were an SpM which could made remote SvM calls. This approach is applicable only if the interface hardware is connected to a network channel to which the node hosting a server TMO is also connected.

### RC4: A special SvM

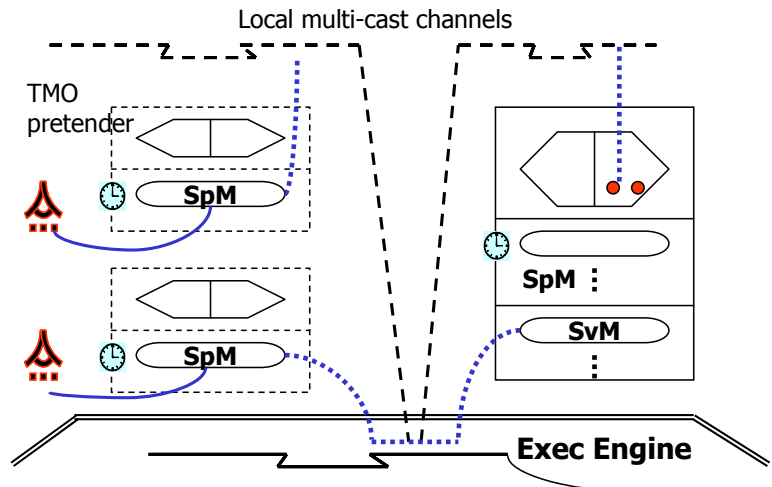


Figure 6. TMO-pretender interfaces to peripherals

This approach is to write an SvM which can be registered with TMOSM so that it may be invoked upon occurrence of a specified signal from the underlying OS kernel. Figure 7 depicts this approach. As a commercial vendor introduces a new peripheral, the peripheral vendor and the OS kernel vendor together add a corresponding API to the APIs provided by the commercial OS kernel. In principle, the newly added API can include hooks through which certain event / signal handlers to run in the user mode can be linked. Then, when a specified condition in a peripheral arises, a linked handler is invoked by the OS kernel. A style of writing such handlers within the TMO framework is to write them in the form of SvMs. Such SvMs are "special SvMs" in that they are triggered not by service request messages coming from other TMOs but instead by signals coming from local peripherals through the local TMO execution engine. However, most of the current popular OS kernels do not appear to provide I/O APIs through which signals produced inside the kernel can directly lead to invocation of functions in the user mode.

In the case of the Microsoft Windows OS family, an interrupt signal from a peripheral is first handled by an interrupt service routine (ISR). The ISR can schedule deferred procedure calls (DPCs) so that the functions corresponding to DPCs will be executed some time after completion of the ISR and successful competition against other computation units for getting the CPU resources. Therefore, in principle, target functions of DPCs can be written to invoke high-level handlers of the signals from peripherals in the user mode. However, ISRs and DPC target functions of most commonly used peripherals do

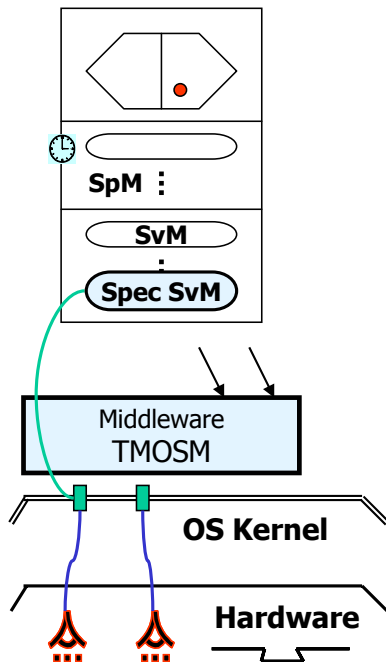


Figure 7. Special SvM registered to respond to signals from peripherals

not seem to provide such hooks. This means that if one wants to facilitate reactive control of a peripheral by using a special SvM, the person needs to modify the ISR to schedule a DPC for a function that will invoke a function in TMOSM, where the latter function will in turn invoke a special SvM.

Therefore, the special SvM approach is not broadly applicable. In general, a TMOSM implementation is tailored to the adopted OS kernel. Only if the I/O API provided by the OS kernel for a certain type of peripherals includes the kinds of hooks discussed above, then a TMOSM can be implemented to possess the capability of conveying signals from the kernel part dealing with those peripherals to pre-registered special SvMs. Otherwise, the ISRs of peripherals need to be extended to incorporate calls for certain functions in TMOSM in order to facilitate the special SvM approach.

## 5. Conclusion

Several basic styles of writing TMO parts for controlling I/O activities including reactive control of peripherals have been devised with the goal of establishing general and efficient styles and enabling relatively easy analysis of the timing behavior of such TMOs. These I/O programming styles were discussed in this paper. All the approaches except the two, RC3 and RC4, have been used in TMO programming experiments so far. All the approaches are of practical nature although they have somewhat different application ranges. Further in-depth study of performance aspects of these different approaches is desirable.

**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 (NEST Program) 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.

## References

- [Bol00] Greg Bollella and James Gosling, "The Real-Time Specification for Java", *Computer*, June, 2000, pp. 47-54.
- [IEE00] 'A special issue of Computer (a magazine of IEEE Computer Society) on Object-oriented Real-time distributed Computing', June 2000.
- [Kim97] Kim, K.H., "Object Structures for Real-Time Systems and Simulators", *IEEE Computer*, August 1997, pp.62-70.
- [Kim99] Kim, K.H., Ishida, M., and Liu, J., "An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-based Implementation", *Proc. ISORC '99 (IEEE CS 2nd Int'l Symp. on Object-oriented Real-time distributed*

Computing), May 1999, pp.54-63.

[Kim00a] Kim, K.H., "APIs for 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, pp.10-20, (Keynote paper).

[Kim01] Kim, K.H., Liu, J.Q., Miyazaki, H., and Shokri, E.H., "TMOES: A CORBA Service Middleware Enabling High-Level Real-Time Object Programming", *Proc. ISADS 2001 (IEEE CS 5th Int'l Symp. on Autonomous Decentralized Systems)*, Dallas, TX, March 2001, p. 327-335.

[Kop97] Kopetz, H., '*Real-Time Systems: Design Principles for Distributed Embedded Applications*', Kluwer Academic Publishers, ISBN: 0-7923-9894-7, Boston, 1997.

[OMG00a] 'Collection of Slide Presentations, 1<sup>st</sup> OMG Workshop on Real-Time / Embedded Distributed Object Computing', July 2000, Crystal City, VA.

[OMG00b] Object Management Group, *The common Object Request Broker: Architecture and Specification*, Revision 2.4, Oct, 2000.

[Sch00] Douglas Schmidt, Fred Kuhns, "An Overview of the Real-Time CORBA Specification", *Computer*, June, 2000, pp. 56-63.

[Sel00] Bran Selic, "A Generic Framework for Modeling Resources with UML", *Computer*, June, 2000, pp. 64-69.

# Proceedings

---

**Fifth IEEE International Symposium on  
Object-Oriented Real-Time Distributed Computing (ISORC 2002)**

---

**29 April — 1 May 2002 ♦ Washington, D.C.**

**Editors**

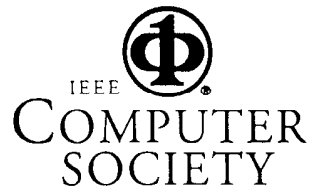
**Luiz Bacellar  
Peter Puschner  
Seongsoo Hong**

***Sponsored by***

IEEE Computer Society Technical Committee on Distributed Processing

***In cooperation with***

OMG and IFIP WG 10.4



Los Alamitos, California

Washington • Brussels • Tokyo

---