An OSEK/VDX Implementation of Synchronous Reactive Semantics Preserving Communication Protococls

Guoqiang Wang
Marco Di Natale
Alberto L. Sangiovanni-Vincentelli

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-81
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-81.html

June 11, 2007

Copyright © 2007, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This research is partly funded by MARCO through the Gigascale Systems Research Center and by the Center for Hybrid and Embedded Software Systems, which receives support from NSF, the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, General Motors, Hewlett Packard, Microsoft, National Instruments and Toyota.

An OSEK/VDX Implementation of Synchronous Reactive Semantics Preserving Communication Protococls*

Guoqiang Wang[1] geraldw@eecs.berkeley.edu
Marco Di Natale[2] marco.dinatale@gm.com
Alberto Sangiovanni-Vincentelli[1] alberto@eecs.berkeley.edu
[1] University of California at Berkeley, CA, USA
[2] General Motors R&D, Warren, MI, USA

05/31/2007

Abstract

Synchronous Reactive semantics preserving communication buffer sizing mechanisms and buffer indexing protocols are presented for both single-port and multi-port tasks. Because these protocols define buffer indices for writers and readers at task activation time, generally they require a kernel-level implementation. In this paper, we present portable implementations for applications with SR semantics under the OSEK OS standard, which is widely used in automotive designs. To meet the one-alarm minimum requirement, an task called `dispatcher` *is constructed to activate all other application tasks. For the CTDBP, the hook mechanism is used to gain atomicity of the termination code for lower-priority readers. Complexities in terms of run time, memory, and implementation are compared for different versions of implementations of the SR semantics preserving protocols.*

1 Introduction

Model-based development of embedded real-time software aims at improving quality by fostering reuse and supporting high level modeling and simulation tools. Synchronous Reactive (SR) models have been traditionally used in the design of hardware logic and more recently for modeling control algorithms and control-dominated embedded applications. Synchronous reactive zero-time semantics is very popular because of the availability of tools for simulation and formal verification of system properties. When implementing a high-level model into code, it is often important to preserve the semantics of its model of computation so to retain the results of the validation and verification results. Preserving the semantics of the model may be a non-trivial task. There are two options to implement an SR multirate model. A single task implementation executes at the base rate of the system. Such an implementation is easier to construct, but often characterized by poor resource utilization. On the other hand, a multi-task implementation can be used, with typically one task for each execution rate, and possibly more. Multi-task implementations allow for a much better schedulability of the resources, but because of the possible preemption, communication may have integrity or non-determinism problem and the implementation raises issues with respect to the preservation of the zero time execution behavior. In this paper, we focus on time-critical applications, modeled on the functional side as a set of SR tasks. On the architecture side, single-processor execution platforms with priority-based preemptive scheduling of tasks are the implementation target.

Any (real-time) data communication between concurrent tasks that cannot be made atomic at the hardware level must be implemented using one of the following three communication schemes: *lock-based*, *lock-free*, and *wait-free*. A lock-based scheme is also know as a blocking mechanism. Under such a scheme, when a task wants to access the shared communication data while another task holds an exclusive lock, it blocks (usually on a semaphore), and releases the CPU. When the lock is released, the task is restored in the ready state and can safely access the data. Both lock-free and wait-free schemes are a non-blocking mechanism. Under a lock-free scheme, when a reader wants to access the shared communication data, it does so without blocking. At the end of the reading operation, it performs a validity check on the data. If realizing there

*This research is partly funded by MARCO through the Gigascale Systems Research Center and by the Center for Hybrid and Embedded Software Systems, which receives support from NSF, the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, General Motors, Hewlett Packard, Microsoft, National Instruments and Toyota.

was a possible concurrent operation by the writer and the possibility of having read an inconsistent value, then it performs the read operation again. Leveraging the timing properties of tasks, the number of retries can be upper bounded. Under a wait-free scheme, both writer and its readers are protected against concurrent access by replicating the communication buffers and by leveraging buffer accessing time instants and scheduling constrains such as task priorities and periods.

The wait-free communication scheme has been traditionally researched from the perspective of the programmers of concurrent real-time applications, interested in preserving the consistency of the data and providing the *execution-time freshest value*, meaning that each reader always obtains the latest data written by the writer into the channel. However these protocols cannot guarantee time determinism. Meanwhile, the wait-free scheme has also been the preferred choice to implement semantics-preserving communication protocols due to their simplicity and efficiency. In the following paragraphs, we review some typical wait-free protocols that preserve either the execution-time freshest value semantics or the SR semantics. We start with communication between a single write and a single reader.

In [1], a three-slot asynchronous protocol is proposed to preserve data consistency with execution-time freshest value semantics for the communication between a single writer and a single reader running on a shared-memory multiprocessor. No assumption is made on task priorities and periods. In general three buffers are needed: one for the data being read, one for the data last written (current) by the writer, and another when the latest written buffer has not been read yet, but the writer wants to write a new data item. To achieve data integrity, a hardware-supported Compare-And-Swap (CAS, or another equivalent) instruction needs to be used to atomically assign the reading position in the buffer array to reader tasks and to update the pointer to the last written value.

A one-to-one communication mechanism that preserves the SR semantics has been presented in [2]. A two-place buffer, two buffer indices, and a reader execution flag are required. In the case of single processor systems, given that the code that updates the index variables is executed inside the kernel, at task activation time, there is no need for a CAS instruction, or any other mechanism that ensures atomicity when swapping buffer pointers or comparing state variables. To guarantee that in the low to high priority communication with exactly one unit delay, the SR semantics preserving communication mechanism in [2] requires that each writer task instance completes before the next is activated.

In the general case of multiple reader tasks, wait-free schemes can be constructed by leveraging two properties of the relationship between the writer and its reader tasks. The first method consists in preventing concurrent accesses by computing an *upper bound for the maximum number of buffers that can be used at any given time by reader tasks.* In the worst case, when no additional information is available, this is equal to the maximum number of reader task instances that can be active at any time (the number of reader tasks if task deadlines are less than or equal to periods), plus two more buffers that must be added for guaranteeing that the writer can safely update the latest data. This bound has been defined in [3], where the protocol in [1] was extended to an asynchronous protocol for single-writer and multiple-reader systems. This protocol needs `NR+2` buffer slots and `NR+1` control variables, where `NR` is the number of readers in the system. N is the maximum number of buffers that are in use by the reader tasks at any time. As in the one-to-one communication case, two more buffers need to be added, one for the data being written by the writer and the other when the latest written buffer has not been read yet by any task, but the writer wants to write new data.

In [4] an SR semantics preserving protocol is provided and the buffer bound has been further extended for the case of communication links with a unit delay under the assumption that each task instance terminates before its next activation event. The proposed protocol is called DBP (Dynamic Buffering Protocol). When unit delays are allowed on links, `NLPR+2` buffers are still demonstrably sufficient, where `NLPR` is the number of readers that have a lower priority than the writer.

The other method provides buffer sizing and access procedures by using *temporal concurrency control* that ensures writer and reader tasks never access the same data item at the same time. The size of the buffer can be computed by upper bounding the number of times the writer can produce new data items while a given data item is considered valid by at least one reader. This concept has been first introduced (together with a lock-free protocol implementation) in [5] and [6], assuming as the validity time of the data the worst case execution time of a reader. In [5], a timing-based wait-free mechanism called asynchronous circular buffering protocol is proposed to preserve the execution-time freshest value semantics for a single-writer multiple-reader system on a shared-memory multiprocessor platform with a single global clock. Data sharing is implemented through a sequential algorithm using a circular buffer. In [6] the Non-Blocking Write (NBW) protocol is presented for a single-writer multiple-reader system executing under a priority-based preemptive scheduling on a distributed real-time system consisting of a set of nodes connected by a broadcast communication chan-

nel. Access to the communication channel is assumed through Time Division Multiple Access (TDMA).

The temporal concurrency control concept is also used in [7] for buffer sizing while preserving the SR semantics. An upper bound on the buffer size is based on the data validity lifetime.

In an SR semantics-preserving implementation, we need to ensure that the reader accesses the data produced by the correct instance of a writer task. In particular, the buffer slot that contains the item produced by the writer has to be defined at the writer activation time. Similarly, the buffer item read by a reader is defined at the reader activation time.

Later, at application execution time, the writer and the reader will use the buffer positions defined at their activation time. Task priorities and deadlines ensure that the reader reads the data produced by the correct writer instance. Of course, there may be cases in which the writer produces multiple outputs before the reader completes its execution. In this case, the implementation must necessarily consist of an array of buffer entries in which pointers (indices) are assigned to the writers and the readers to find the right slot in the data structures. Both writer and reader tasks, however, are not guaranteed to start their execution at their release time because of scheduling delays. Therefore, in general, the selection of the data buffer entry that will be written into or read from must be delegated to the operating system (or to a hook procedure that is guaranteed to be executed at the task activation time).

In this paper, we present OSEK/VDX implementations of SR semantics preserving communication protocols. OSEK/VDX is a series of industrial standards particularly for automotive designs. The standards include Operating System (OS), Communication (Com), network management (NM), and debugging (ORTI). For our purpose, we are only interested in the OSEK OS [8] and the accompanying development language.

OSEK OS has already been more than a standard. There exist OSEK design libraries that provide kernel service primitives. OSEK supports a modular design of Real-Time Operating Systems (RTOS). There exist two levels of design reuse: application reuse and RTOS kernel service primitive reuse. To gain implementation automation, OSEK has also developed its own Implementation Language (OIL) [9]. OIL is a mechanism used to statically configure the OS objects required in an OSEK application implementation. An OIL configuration consists of an implementation definition and an application definition, which can be in a single or multiple OIL files. The implementation definition is provided by RTOS vendors and designers only need to prepare their application definition files. An OIL configuration file can be coded manually or generated automatically by design tools. A tool called System Generator (SG) provided by RTOS vendors takes as input the OIL configuration files, automatically selects required kernel services, and produces additional supporting application code. Finally the application source code directly from designers, the primitive files from the OSEK library, and those produced by the SG are compiled and linked together to generate an executable OSEK application file.

Note that the rate transition buffering scheme used by the Real-Time Workshop tool from Mathworks preserves the SR semantics at application execution level. The rate transition buffering scheme is defined for a single writer to single reader communication, where the communicating tasks must have harmonic periods and must be activated with the same phase [10].

Our main contribution of this work is that this is the first implementation of inter-task communication protocols with kernel-level support to preserve the SR semantics. We generalized protocols for single-port tasks to deal with systems with multiple-port tasks and presented the details of buffer sizing mechanisms and indexing procedures at the imperative code level with all the required data structure support. In addition, we analyzed the tradeoffs between different versions of implementations in terms of time, space, and implementation complexity.

The paper is structured as follows: following the introduction section, definitions and notations are presented in Section 2. Then SR semantics preserving communication buffer sizing mechanisms and indexing procedures are presented in Sections 3 and 4. Next, basics of OSEK/VDX and the OSEK application design process are introduced in Section 5. Implementations of the SR semantics preserving protocols under the OSEK OS standard are presented in Section 6. The paper is concluded with conclusions and future work in Section 7.

2 Definitions and Inter-Task Communication Model

We define a task, denoted by τ, as a piece of code that communicates with its environment (other tasks) using ports. Ports can be either input or output and tasks can have either one or more ports. Figure 1 shows a task with `NIP` input ports and `NOP` output ports.

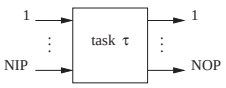

Figure 1. A Task with Multiple Input and Output Ports

Task τ_i is characterized by a set of parameters: priority π_i, period (periodic or sporadic task) T_i, the worst case computation time C_i, the worst case response time R_i, and the relative deadline d_i. Schedulability of tasks

requires that $R_i \leq d_i$.

In our study, the worst case response time of task i can be either equal to or smaller/larger than its period. Therefore there may exist multiple active instances of a task in the system. The instances of a task are executed in a first-in-first-out order. Let $a_i(j)$ be the activation time of the j^{th} instance of τ_i. Under the SR semantics, the activation time $a_i(j)$ captures also the start time and the finish time of the same instance of task τ_i.

We assume that a task only write and read once through its corresponding input ports and output ports, respectively. There is no assumption on when the writes and reads may occur. We further assume that all ports of a task inherit all its temporal properties. Specifically, all ports have the same sampling rate, priority, relative deadline, and activation time as their owner task.

Depending on whether having an input or output port, a task can be either a reader, a writer, or both. A reader and a writer correspond to an input port and an output port, respectively. When a task has a single port, we also call the task a reader or a writer task.

Let w and r_i denote a writer and its reader i, respectively. The owner task of w is denoted by τ_w. Similarly, the owner task of r_i is denoted by τ_{r_i}. Let $out_w(j)$ be the value associated with the j^{th} instance of w and $in_{r_i}(j)$ be the value associated with the j^{th} instance of r_i. We introduce λ to denote the number of active instances of a writer, i.e.

$$\lambda = \left\lceil \frac{R_w}{T_w} \right\rceil.$$

We define $\zeta_i(t)$ to be the number of times that i has occurred up to time t, i.e.

$$\zeta_i(t) = \sup\{m | a_i(m) \leq t\},$$

where i can be a reader or a writer. Note that the sup of an empty set is defined to be zero. We further introduce the offset, denoted by o_{wi}, between $r_i(k)$ and its writer $w(j)$, i.e.

$$o_{wi} = a_i(k) - a_w(j),$$

where $j = \sup\{m | a_w(m) \leq a_i(k)\}$. By definition, O_{wi}, the worst case value of o_{wi}, is smaller than the period of the writer, i.e. $O_{wi} < T_w$.

Communication is defined between a writer and all its readers. We allow delays along communication links. Let `delay[i]` represent the link delay for reader i. Communication link delays are design parameters. However for readers with a higher priority, a legitimate link delay must be at least λ.

Figure 2 shows the general case of communication between one writer and its NHPR+NLPR readers, among which NHPR readers have higher priorities while NLPR readers have lower priorities compared with the writer. The non-positive numbers associated on the arcs represent link delays. For example, among the NLPR lower-priority readers, N_0 readers communicate with the writer with no delay, N_1 readers with a unit delay, etc. Similarly, among the NHPR higher-priority readers, M_0 readers communicate with the writer with λ-unit delays, M_1 readers communicate with $(\lambda+1)$-unit delays, etc. Let κ represent the longest link delay associated with communication links, i.e. $\kappa = \max(p, \lambda + q)$.

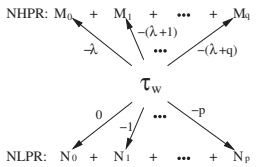

Figure 2. General Case Communication between One Writer and Multiple Readers

Therefore the SR communication semantics can be mathematically formulated as follows:

$$in_{r_i}(j) = out_w(k),$$

where $k = \max\{0, \zeta_w(a_i(j)) - delay[i]\}$.

The top of Figure 3 illustrates the execution of a pair of tasks communicating with the SR zero-time semantics. The horizontal axis represents time. The vertical arrows capture the time instants when the tasks are activated and compute their output from the input values. Note that, in the middle of the figure, it is $i_{r_i}(j) = o_w(k)$ during simulation. The bottom of Figure 3

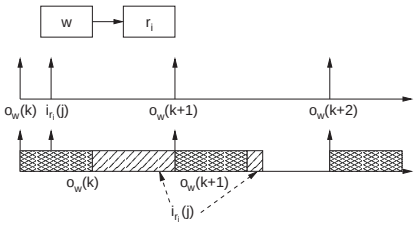

Figure 3. Data Integrity and Determinism Problems due to Preemption

shows the possible problems with data transfers in a multi-task implementation when buffers are addressed at execution time. A fast writer, implemented by a high priority task, communicates with a slow reader. The writer finishes its execution producing output $o_w(k)$ and the reader is executed right after. If the reader performs its read operation before the preemption by the next writer instance, then $i_{r_i}(j) = o_w(k)$. Otherwise, it is preempted and a new instance of the writer produces $o_w(k+1)$. In case the read operation had not been performed before, the task reads $o_w(k+1)$, in general different from the value $o_w(k)$. Even worse, in case the data item is not read atomically, there is a finite probability that $w(k)$ preempts the reader task r_i while a read is in progress, resulting in an inconsistent value and a data integrity problem.

In the following sections, SR semantics preserving communication buffering sizing mechanisms and protocols are presented for systems with the inter-task communication model as shown in Figure 2.

3 SR Semantics Preserving Buffer Sizing Mechanisms

Any communication scheme consists of two parts: a buffer sizing mechanism and a buffering indexing procedure. In this section, we first present two mechanisms used to size communication buffers and in Section 4 we present the corresponding buffer indexing procedures.

3.1 Based on Spatially-out-of-Order Writes

The first mechanism used to size a communication buffer, as shown in Figure 4, is based on the active number of reader instances of a writer. The writer and its readers share array `Buf[]` for data communication. Since the writer writes data into the buffer in a spatially non-sequential manner, it is also called a mechanism based on spatially-out-of-order writes.

Similar to the unit delay case presented in [4], we need to keep only one copy of the current and the previous κ buffer indices. On the writer side, a circular array, $\texttt{pos}[\kappa+1]$, fulfills this purpose. When a new instance of the writer task is activated, the old buffer index with κ-unit delay becomes the new buffer index with $(\kappa+1)$-unit delays, which is not needed and therefore used for storing the new current buffer index. Similarly the old current buffer index becomes the new buffer index with a unit delay. Integer variable `cur` is used to index the entry in `pos[]` that stores the current buffer index.

On the readers' side, array `Read[]` stores the buffer index used by all the reader instances. Note that the instances of a reader are stored in a contiguous subset of `Read[]`. Similar to the array `pos[]`, each subset of `Read[]` for a reader is used as a circular array. The array `Ri[]` is used to index the currently executing instance of a reader. Note that `Ri[j]` and its corresponding contiguous subset in `Read[]` function similarly to `cur` and `pos[]`. Though we present the array `pos[]` with the notion of link delays, however it also embeds the notion of multiple writer instances.

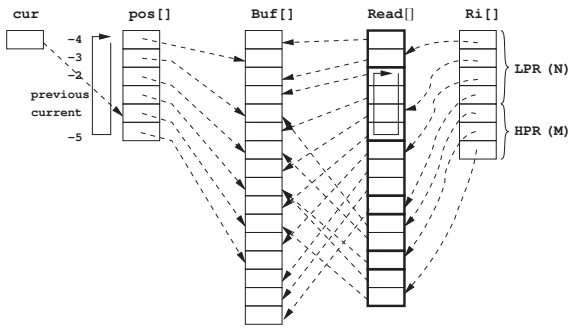

Figure 4. Based on Spatially-out-of-Order Writes

The total number of instances of lower-priority readers is computed as follows:

$$\texttt{ILPR} = \sum_{\texttt{j}\in\texttt{lp(w)}} \left\lceil \frac{\texttt{R}_{\tau_{\texttt{r}_\texttt{j}}}}{\texttt{T}_\texttt{j}} \right\rceil,$$

where `lp(w)` represents the readers with a lower priority than the writer. Assuming all tasks have unique priorities, the worst case response time can be computed according to the schedulability theory:

$$\texttt{R}_{\tau_{\texttt{r}_\texttt{i}}} = \texttt{C}_{\tau_{\texttt{r}_\texttt{i}}} + \sum_{\texttt{j}\in\texttt{hp(i)}} \left\lceil \frac{\texttt{R}_{\tau_{\texttt{r}_\texttt{i}}}}{\texttt{T}_\texttt{j}} \right\rceil \texttt{C}_\texttt{j},$$

where `hp(i)` denotes the set of tasks that has a higher priority than that of reader `i`.

Some of the entries in arrays `pos[]` and `Read[]` may certainly share some buffer indices, but in the worst case, all of them may index unique buffer entries. Therefore the size of the buffer can be computed as follows:

$$\texttt{NB} = \texttt{ILPR} + \kappa + 1, \tag{1}$$

among which `ILPR` slots are reserved for lower-priority reader instances, κ slots store the writer outputs with a delay from unity to κ units, and one entry is for the writer to write into a new data item. Note that all the higher-priority readers share the same copy of the communication data with a certain communication link delay. This mechanism is also called a buffer sizing mechanism based on the number of instances of low-priority reader.

Up to now, the buffer has been sized without using temporal properties between the writer and readers. Actually the buffer size can be improved because only the minimum between the number of active reader instances and the number of writes is needed during the worst case execution time of a reader. Therefore the bound on the buffer size can be improved as follows:

$$\texttt{NB} = \sum_{\texttt{j}\in\texttt{lp(w)}} \left(\min \left(\left\lceil \frac{\texttt{R}_{\tau_{\texttt{r}_\texttt{j}}}}{\texttt{T}_\texttt{j}} \right\rceil, \left\lceil \frac{\texttt{R}_{\tau_{\texttt{r}_\texttt{j}}}}{\texttt{T}_\texttt{w}} \right\rceil \right) \right) + \kappa + 1. \tag{2}$$

3.2 Based on Spatially-in-Order Writes

The other mechanism used for communication buffer sizing allows a writer to write data into a buffer in a spatially sequential order. In short, we call it mechanism based on spatially-in-order writes.

Assume that some writer instance `k` happens at time $\texttt{a}_\texttt{w}(\texttt{k})$ and the writer updates a buffer position of index `n` as shown in Figure 5. The item in position `n` is used by the readers that are activated during the time interval of $[\texttt{a}_\texttt{w}(\texttt{k}+\texttt{delay[i]}), \texttt{a}_\texttt{w}(\texttt{k}+\texttt{delay[i]}+1))$ and use a communication link with a $\texttt{delay[i]}$-unit delay.

The buffer slot indexed by `n` must remain valid until any reader activated in these intervals has finished its execution. Future instances of the writer use buffer slots with indices `n+1`, `n+2`, and so on, until, eventually, the buffer index wraps around the circular buffer and goes back to position `n-1`. The condition for a correct buffer sizing is that all the reader instances that used the previous buffer at position `n` finished using the data when some future writer instance goes back to position `n` and overwrites it.

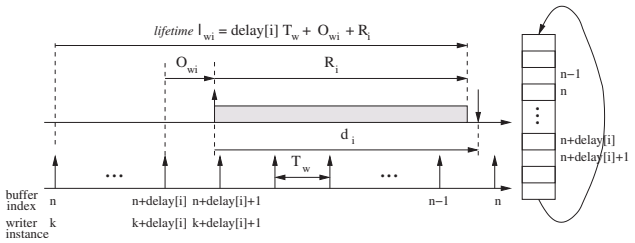

Figure 5. Based on Spatially-In-Order Writes

Figure 5 illustrates the basic idea about this mechanism. We define the maximum lifetime of the data produced by the writer for the reader r_i, denoted by l_i, as follows:

$$l_i = \texttt{delay[i]} \times T_w + O_{wi} + R_i.$$

Let NR be the number of readers of a writer and therefore the buffer size can be computed as follows:

$$\texttt{NB} = \max_{1 \le i \le \texttt{NR}} \left\lceil \frac{l_i}{T_w} \right\rceil. \quad (3)$$

The writer just keeps writing data into the next slot in a circular buffer at its own sampling rate. The good thing is that this mechanism does not require much bookkeeping to achieve constant execution time for finding a safe buffer slot for the writer. It is also called Temporal Concurrency Control (TCC) based buffer sizing mechanism because it mainly relies on the temporal properties of a writer and its readers.

4 SR Semantics Preserving Communication Protocols

In this section we present SR semantics preserving communication protocols for tasks whose deadlines are not greater than their periods. This implies that only one active instance for each task exists in the system at any time. We only consider communication links with a maximum of one unit delay. Therefore for readers with a higher priority, the link delay must be unity, while for readers with a lower priority, the link delay is a design parameter that can be either zero or one. With the above assumption, λ, p, and q in Figure 2 are 1, 1, and 0, respectively. Furthermore, the data structures shown in Figure 4 can be simplified since pos[] and cur degenerate to a pair of variables (prev, cur). Similarly Ri[] and Read[] degenerate to a single array Read[].

In the rest of this section, we present the SR semantics preserving inter-task communication protocols with buffer sizing based on either spatial-out-of-order writes or spatial-in-order writes. We analyze and compare their complexity in terms of time and space. To simplify our discussion, we first present the version of the protocols for systems with single-port tasks and then extend them for systems with multiple-port tasks.

4.1 Protocols for Single-Port Tasks

For convenience, notations are first summarized in Table 1.

cur/prev	buffer slot with latest/immediate previous data
NLPR	number of lower-priority readers
NR	number of readers
NT	number of tasks
WrtInit	initial output value of writer
Buf[NB]	communication buffer with NB slots
delay[i]	communication link delay for reader i
Read[i]	buffer slot currently used by reader i

Table 1. Notations Used to Describe a System

4.1.1 The Dynamic Buffering Protocol

The high-level pseudo-code of the Dynamic Buffering Protocol (DBP) for single-writer multiple-reader systems is shown in Figure 6, as defined in [4]. The code takes advantage of the fact that buffer index updates are performed by the kernel at task activation time and therefore, in single-processor execution platforms, they are atomic for both the writer and the reader tasks. There are different ways to implement the FindFree() search algorithm used to find a safe buffer slot for the writer at its activation time. We present two versions of implementation. Because data structures need to be adjusted for different versions, there may be possible repetitions in the full version of their code implementation.

Data Structures

```
char cur, prev;                 char Read[NLPR];
message Buf[NB];                char HPR[NHPR];
```

Writer

activation time	*execution time*
prev = cur;	···
cur = FindFree();	Buf[cur] = ···
	···

```
FindFree() {
  return j∈[1,NLPR+2]if prev≠j∧∀i∈[1,NLPR] Read[i]≠j;
}
```

Lower Priority Reader

activation time	*execution time*
if (delay[i])	···
Read[i] = prev;	··· = Buf[Read[i]];
else	···
Read[i] = cur;	Read[i]=FREE;

Higher priority Reader

activation time	*execution time*
HPR[i] = prev;	···
	··· = Buf[HPR[i]];
	···

Figure 6. Code for Writer/Readers in [4]

The mechanism based on spatially-out-of-order writes is used for buffer sizing in the DBP. Therefore, as discussed in Section 3.1, the DBP buffer sizing gives a total buffer size of NB = NLPR+2 for a writer. In the following paragraphs, we present the implementations for the DBP with a linear and a constant time search algorithm.

LTDBP: DBP with a Linear Time FindFree() Figure 7 shows the data structures used in the LTDBP. The task descriptor has two fields for each task in the system. For implementation convenience, we

use a single index array, `Read[]`, for both higher and lower-priority readers, which is slightly different from what is in [11], where the DBP is originally presented. We use the portion of `Read[]` with lower indices for lower-priority readers to meet the requirement that the `FindFreeSL()` needs the buffer access indexing array for lower-priority readers to be contiguous.

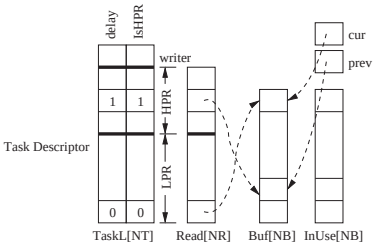

Figure 7. Data Structure for LTDBP (SPT)

Data Structure	Initialization
`char cur, prev;`	`cur = prev = 0;`
`char Read[NR], InUse[NB];`	`Buf[0] = WrtInit;`
`char IsHPR[NR], delay[NR];`	`for (j = 0; j < NLPR; j++)`
`message Buf[NB]; ;`	`  Read[j] = -1;`
Writer Task	
`/* activation time */`	`/* execution time */`
`prev = cur;`	`...`
`cur = FindFreeSL();`	`Buf[cur] = ...`
	`...`
Reader Task i	Def of FindFreeSL(), O(NB)
`/* activation time */`	`char FindFreeSL(void) {`
`if (delay[i])`	`  for (j = 0; j < NB; j++)`
`  Read[i] = prev;`	`    InUse[j] = 0;`
`else`	`  InUse[prev] = 1;`
`  Read[i] = cur;`	`  for (j=0; j<NLPR; j++) {`
`/* execution time */`	`    if (Read[j] != -1)`
`...`	`      InUse[Read[j]] = 1;`
`... = Buf[Read[i]];`	`  }`
`...`	`  for (j=0; InUse[j]; j++)`
`/* termination time */`	`    ;`
`if (IsHPR[i] == 0)`	`  return j;`
`  Read[i] = -1;`	`}`

Figure 8. C Code for the LTDBP (SPT)

Figure 8 shows the required initialization and the buffer indexing procedure of the LTDBP for single-writer multiple-reader systems. It is clear that the communication protocol is executed at two levels: upon activation by the kernel and during execution by the application tasks. Since a linear time `FindFreeSL()` function is used, the execution time of the writer at the kernel level depends in the worst case on the buffer size. The indexing code for readers that needs to be executed at the kernel level clearly finishes in constant time for each reader and the total execution time at the kernel level depends on the number of readers that need to be activated.

Note that right before a lower-priority reader finishes execution, it flags its completion on reading its buffer slot by setting its `Read` value to `-1`. `Read[]` is shared and both the writer and the lower-priority reader may update the same slot concurrently. Since single memory operation is atomic, mutual exclusive access to shared memory is guaranteed. Unlike lower-priority readers, when a higher-priority reader finishes, it does not need to flag its completion because of its higher priority than the writer. The memory consumption for the LTDBP is summarized in Table 2.

variable	char	message
count	3×NR+NB+2	NB

Table 2. LTDBP Memory Requirement (SPT)

CTDBP: DBP with a Constant Time `FindFree()`

Note that the `FindFree()` procedure is executed at the activation time of the writer, which is at the kernel level. A long execution time at the kernel level may force tasks with short deadlines to be unschedulable. Therefore we are particularly interested in the most efficient implementations even with a higher spatial cost.

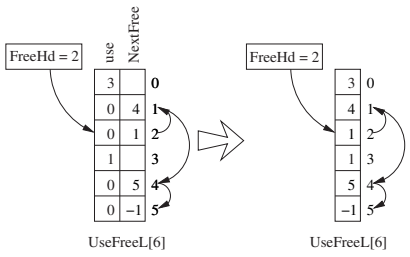

Figure 9. A Use Free List Data Structure

The most efficient implementation for the `FindFree()` algorithm takes constant time, which can be obtained by using a special data structure called use free list as shown in Figure 9. Usually a use-free-list entry contains two fields: use count (`use`) and next free slot index (`NextFree`). The beginning of the free list is indicated by `FreeHd` while the end of the free list is denoted by a value of -1 in the `NextFree` field. In the example shown in Figure 9, the length of the list is six and two entries (0 and 3) are currently used. Indicated by `FreeHd`, the free list starts with entry 2, then goes to entries 1, 4, and 5 in series. `UseFreeL[5].NextFree` is `-1`, which indicates the end of the free list. It is not difficult to observe that the values of the `use` fields along the free list are all zeros. Therefore, to save memory, the two fields can be compacted into one for each entry with the value being the next free slot index if on the free list or the use count otherwise. It is clear that the free entry can be obtained by getting the current `FreeHd` value in constant time, which supports a constant time search algorithm, `FindFreeSC()`.

Figure 10 illustrates the data structures for the CT-DBP. Comparing with Figure 7, field `FreeHd` is added. Also note that lower-priority readers are not necessarily mapped onto a contiguous section in `Read[]`.

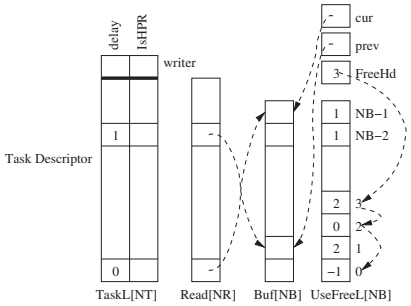

Figure 10. Data Structure for CTDBP (SPT)

Figure 11 shows the corresponding initialization and the indexing procedure. Note that an initial free list is constructed in the initialization phase. During execution, the `FindFreeSC()` returns the current `FreeHd` after assigning the index of the second entry on the free list as the new `FreeHd`. If the current `UseFreeL[FreeHd]` is -1, it simply implies that currently only one entry is free. Under the DBP buffer sizing mechanism, it is guaranteed that the current `FreeHd` is never -1.

Comparing the code for the writer and readers with their counterparts in Figure 8, besides defining the accessing buffer index at activation time, `UseFreeL[]` and `FreeHd` need to be updated during the kernel-level execution for the writer and lower-priority readers. On termination, lower-priority readers update the `UseFreeL[]` and possibly `FreeHd` by a series of load and store memory operations. Unfortunately, this series is not atomic. Any correct implementation must treat them as a critical section. The memory consumption for the CTDBP is summarized in Table 3.

variable	char	message
count	3×NR+NB+3	NB

Table 3. CTDBP Memory Requirement (SPT)

4.1.2 Temporal Concurrency Control Protocol

The mechanism based on spatially-in-order writes is used for the buffer sizing in the Temporal Concurrency Control Protocol (TCCP). Though we are still using `NB` as the buffer size, it may be very different from the buffer size obtained in Section 4.1.1.

The data structures used for the TCCP for single-writer multiple-reader systems is shown in Figure 12. Comparing with the data structures used for the DBP in Section 4.1.1, the task descriptor simply records the `delay` information for each reader.

Figure 13 shows the required initialization and the indexing procedure of the TCCP. Figure 13 shows that the kernel-level implementation for the writer and a single reader executes in constant time. The execution time at the kernel level for the readers depends on the number of readers that need to be activated. Unlike the DBP, there is no particular bookkeeping operation needed for readers to perform at termination time. Since the writer writes data into a circular buffer in a spatially-in-order manner, the `FindFreeST()` simply increments the `cur`, take the modulus operation with respect to the size of the buffer and then return the remainder as the new `cur`. Table 4 shows the memory requirement of the TCCP.

Data Structure	Initialization

`char cur, prev, FreeHd;`
`char IsHPR[NR], delay[NR];`
`char Read[NR], UseFreeL[NB];`
`message Buf[NB];`
	`cur = prev = 0;`
	`Buf[0] = WrtInit;`
	`UseFreeL[0] = 1;`
	`FreeHd = 1;`
	`for (j=1; j<(NB-1); j++)`
	`  UseFreeL[j] = j + 1;`
	`UseFreeL[NB-1] = -1;`

Writer Task	**Reader Task i**

`/* activation time */`
`UseDec(prev);`
`prev = cur;`
`cur = FindFreeSC();`
`UseFreeL[cur] = 1;`
`/* execution time */`
`...`
`Buf[cur] = ...`
`...`
	`/* activation time */`
	`if (delay[i])`
	`  Read[i] = prev;`
	`else`
	`  Read[i] = cur;`
	`if (IsHPR[i] == 0)`
	`  UseFreeL[Read[i]]++;`
	`/* execution time */`
	`...`
	`... = Buf[Read[i]];`
	`...`
	`/* termination time:CS */`
	`if (IsHPR[i] == 0)`
	`  UseDec(Read[i]);`

Def of UseDec(char j)	

`void UseDec(char j) {`
`  UseFreeL[j]--;`
`  if(UseFreeL[j] == 0) {`
`  UseFreeL[j] = FreeHd;`
`  FreeHd = j;`
`  }`
`}`	

Def of FindFreeSC(), O(1)	

`char FindFreeSC(void) {`
`  tmp = FreeHd;`
`  FreeHd = UseFreeL[tmp];`
	`  return tmp;`
	`}`

Figure 11. C Code for the CTDBP (SPT)

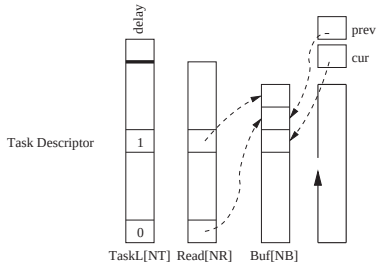

Figure 12. Data Structure for the TCCP (SPT)

Data Structure		Initialization
`char cur, prev;`	`char delay[NR];`	`cur = prev = 0;`
`char Read[NR];`	`message Buf[NB];`	`Buf[0] = WrtInit;`
Writer Task		**Reader Task i**

`/* activation time */`
`prev = cur;`
`cur = FindFreeST();`
`/* execution time */`
`...`
`Buf[cur] = ...`
`...`
		`/* activation time */`
		`if (delay[i])`
		`  Read[i] = prev;`
		`else`
		`  Read[i] = cur;`
		
		`/* execution time */`
		`...`
		`... = Buf[Read[i]];`
		`...`

Def of FindFreeST(), O(1)		
`char FindFreeST()` `{  return (cur+1)%NB;  }`		

Figure 13. C Code for the TCCP (SPT)

variable	char	message
count	2×NR+2	NB

Table 4. TCCP Memory Requirement (SPT)

4.2 Protocols for Multi-Port Tasks

In Section 4.1, protocols are presented for systems with single-port tasks. In reality, systems almost always contain tasks with multiple ports. A task can be both a reader and a writer. In this section, we generalize the protocols presented earlier to cover systems with multiple-port tasks.

A writer in Section 4.1 correspond to an output port of a multiple-port task, respectively. Consequently `cur`, `prev`, and `NLPR` need to be defined for each output port of a task. Similarly initial values need to be provided for each output port of a task in the system.

On top of the notations presented in Table 1, more notations are further introduced to characterize systems with multi-port tasks. Let `SysNIP` and `SysNOP` be the number of input ports and output ports, respectively. Let `SysNB` be the total number of buffers required. `SysNB` is simply the sum of buffer sizes for all writers:

$$\texttt{SysNB} = \sum_{1 \le o \le \texttt{SysNOP}} \texttt{NB}_{\texttt{w}_o}, \tag{4}$$

where $\texttt{NB}_{\texttt{w}_o}$ is computed by using Equation 1 or 2 and Equation 3 for the DBP and TCCP, respectively. Note that $\kappa = 1$ according to our assumptions.

4.2.1 The Dynamic Buffering Protocol

For multi-port tasks, the data structures need to be extended. As the data structures shown in Figures 7 and 10, a task descriptor is needed to capture properties of tasks. But the task descriptor itself is not sufficient any more and thus two more descriptors for input ports and output ports are introduced. All readers and writers share array `Buf[SysNB]`. In the rest of this section, we present the details of the extended versions for the LTDBP and the CTDBP.

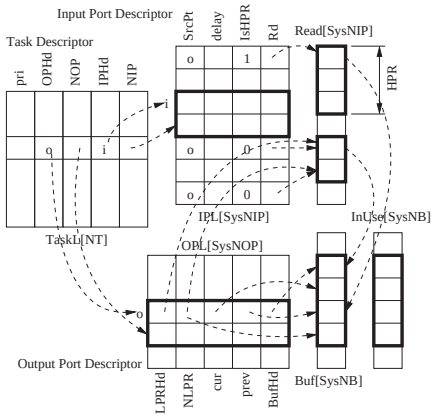

Figure 14. Data Structure for LTDBP (MPT)

LTDBP: DBP with a Linear Time `FindFree()` Figure 14 shows the data structures used for the LTDBP for systems with multi-port tasks. The task descriptor has an entry for each task specifying its input and output port information. The input port descriptor characterizes the properties of each input port (reader) in the system. Specifically, each entry in the descriptor records communication source port (`SrcPt`), link delay, relative priority, and buffer access index (`Rd`). Similarly, the output port descriptor specifies the properties of each output port (writer) in the system. All input and output ports of a task are stored in the respective descriptor consecutively as specified by (`OPHd, NOP`) and (`IPHd, NIP`) in Figure 14. The pair of fields `LPRHd` and `NLPR` specifies a contiguous segment in `Read[]` for all lower-priority readers of a writer. In our implementation, we use the low-index portion of the `Read[]` for lower-priority readers and the high-index portion for all higher-priority readers. Note that the portion of `Read[]` for higher-priority readers of a writer does not have to be contiguous. Similarly, `BufHd` and `NB` specify a contiguous segment in `Buf[]` for a writer.

Data Structure

```
struct TaskEntry {
  char pri;
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT];
```

```
struct OPEntry {
  char LPRHd;
  char NLPR;
  char cur;
  char prev;
  char BufHd;
} OPL[SysNOP];
```

```
struct IPEntry {
  char SrcPt;
  char delay;
  char IsHPR;
  char Rd;
} IPL[SysNIP];
```

```
char Read[SysNIP];
char InUse[SysNB];
```

```
message WrtInit[SysNOP];
message Buf[SysNB];
```

Initialization

```
/* OPL[].BufHd/LPRHd/cur/pre, buffer */
OPL[0].cur=OPL[0].prev=OPL[0].BufHd=OPL[0].LPRHd=0;
Buf[OPL[0].BufHd] = WrtInit[0];
for (i = 1; i < SysNOP; i++) {
  OPL[i].BufHd = OPL[i-1].BufHd + OPL[i-1].NLPR + 2;
  OPL[i].cur = OPL[i].prev = OPL[i].BufHd;
  Buf[OPL[i].BufHd] = WrtInit[i];
  OPL[i].LPRHd = OPL[i-1].LPRHd + OPL[i-1].NLPR;
}
/* IPL[].IsHPR/Rd and LPR Read[] */
char tmp1[SysNIP] = {0, ···};
j = SysNIP - 1;
for (i = 0; i < SysNIP; i++) {
  idx1 = IPL[i].SrcPt;
  idx2 = OPL[idx1].owner;
  if (TaskL[IPL[i].owner].pri > TaskL[idx2].pri) {
    IPL[i].IsHPR = 1;
    IPL[i].Rd = j;
    j = j - 1;
  } else { /* contiguous Read[] for LPR */
    IPL[i].IsHPR = 0;
    IPL[i].Rd = tmp1[idx1] + OPL[idx1].LPRHd;
    tmp1[idx1]++;
  }
  Read[i] = -1;
}
```

Figure 15. DS Initialization for LTDBP (MPT)

Figures 15, 16, and 17 give the complete LTDBP for multi-port tasks. The data structure declaration and the initialization code is shown in Figure 15. Three descriptors are declared as structs and their corresponding initializations are shown at the bottom. Similarly, `cur`, `prev`, and the buffer initial value are initialized for each writer. Note that lower-priority readers of the same writer are mapped to a contiguous segment in array `Read[]` by their `Rd` values.

```
/* activation time */
/* each writer i2 */
OPL[i2].prev = OPL[i2].cur;
OPL[i2].cur = FindFreeML(i2);

/* each reader i2 */
i3 = IPL[i2].SrcPt;
i4 = IPL[i2].Rd;
if (IPL[i2].delay)
  Read[i4] = OPL[i3].prev;
else
  Read[i4] = OPL[i3].cur;
```

```
/* execution time */
...
/* each writer k */
Buf[OPL[k].cur] = ...
...
/* each reader k */
...=Buf[Read[IPL[k].Rd]];
...
/* termination time */
/* each LPR k */
if (IPL[k].IsHPR == 0)
  Read[IPL[k].Rd] = -1;
```

Figure 16. Application Tasks for LTDBP (MPT)

Figure 16 shows the code for application tasks with multiple ports when using the LTDBP . Comparison with Figure 8 reveals that the extended version of LTDBP essentially performs the same functionality, but for all ports (readers and writers) of a task. Since there may exist multiple writers in the system, at activation time of a task, the source communication port (writer) of a reader needs to be identified to get the right copy of `pre` and `cur`.

```
char FindFreeML(char i) {
  char NLPR = OPL[i].NLPR;
  char prev = OPL[i].prev;
  char BufHd=OPL[i].BufHd;
  char LPRHd=OPL[i].LPRHd;
  char NB = NLPR + 2;
  for (k=0; k<NB; k++)
    InUse[k+BufHd] = 0;
  InUse[prev] = 1;
  for (k=0; k<NLPR; k++) {
    j = Read[k+LPRHd];
    if (j != NB)
      InUse[j] = 1;
  }
  for(k=BufHd;InUse[k];k++)
    ;
  return k;
} /* O(NB) */
```

Figure 17. `FindFreeML()` for LTDBP (MPT)

Figure 17 shows the `FindFreeML()` for the extended LTDBP. It is actually almost the same as its counterpart, `FindFreeSL()`, shown in Figure 8. The only difference is that `FindFreeML()` takes as argument the writer's index, which guides the search algorithm to use the data belonging to this writer.

With the extension to handle multiple-port tasks, more memory is needed as shown in Table 5.

variable	char	message
count	5×(NT+SysOP+SysNIP)+SysNB	SysNB+SysNOP

Table 5. LTDBP Memory Requirement (MPT)

CTDBP: DBP with a Constant Time `FindFree()`

In this section, we extend the CTDBP for single-port tasks to handle systems with multiple-port tasks. Figure 18 shows the data structures used for the extension. Notice that the task descriptor is exactly the same as its counterpart in Figure 14.

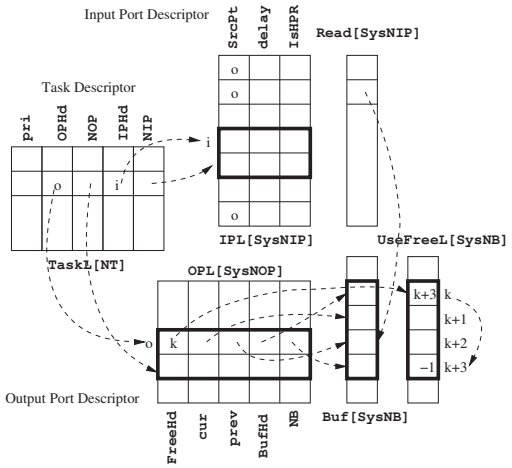

Figure 18. Data Structure for CTDBP (MPT)

Since the `FindFreeMC()` search function only reads information from the use free list, the fields `LPRHd` and `Rd` are not needed any more. All readers use their own indices to index array `Read[]` when referencing buffer slots. Since there is one use free list for each writer in the system, `FreeHd` needs to be defined for all writers as shown in Figure 18.

```
Data Structure
char Read[SysNIP];
char UseFreeL[SysNB];
message Buf[SysNB];

struct IPEntry {
  char SrcPt;
  char delay;
  char IsHPR;
} IPL[SysNIP];

struct OPEntry {
  char FreeHd;
  char cur;
  char prev;
  char BufHd;
  char NB;
} OPL[SysNOP];

message WrtInit[SysNOP];
struct TaskEntry TaskL[NT];

Initialization
... /* same BufHd, IsHPR, & buffer init as in Fig 15 */
/* initialization of use free list*/
for (i = 0; i < SysNOP; i++) {
  UseFreeL[OPL[i].BufHd] = 1; /* not free */
  OPL[i].FreeHd = OPL[i].BufHd + 1;
  for (j = 2; j < (OPL[i].NB-1); j++) {
    k = j + OPL[i].BufHd;
    UseFreeL[k] = k + 1;
  }
  UseFreeL[OPL[i].NB-1+OPL[i].BufHd] = -1;
}
```

Figure 19. DS Initialization for CTDBP (MPT)

Figure 19 shows the data structure declaration and the corresponding initialization. Similar to the LTDBP (MPT) case in Figure 15, the `BufHd` fields are assigned and the corresponding buffer slots are filled with the given initial values for each writer. Then initial unique free lists are built for each writer in the same way as shown in Figure 11. For simplicity, a contiguous buffer segment is associated with a writer.

Similar to the previously presented versions of the

```
/* activation time */
/* each writer i */
UseDecM(i, OPL[i].prev);
OPL[i].prev = OPL[i].cur;
OPL[i].cur=FindFreeMC(i);
UseFreeL[OPL[i].cur] = 1;
/* each reader i */
j = IPL[i].SrcPt;
if (IPL[i].delay)
  Read[i] = OPL[j].prev;
else
  Read[i] = OPL[j].cur;
if (IPL[i].IsHPR == 0)
  UseFreeL[Read[i]]++;
/* execution time */
...
/* each writer k */
Buf[OPL[k].cur] = ···
...
/* each reader k */
··· = Buf[Read[k]];
...
```

```
/* termination time(CS) */
if (IPL[k].IsHPR == 0) {
  t1 = Read[k];
  t2 = IPL[k].SrcPt;
  UseDecM(t2,t1);
}
Def of UseDecM(char i,char j)
void UseDecM(char i,char j){
  UseFreeL[j]--;
  if(UseFreeL[j] == 0) {
    freeHd = OPL[i].FreeHd;
    UseFreeL[j] = freeHd;
    OPL[i].FreeHd = j;
  }
}
Def of FindFreeMC(char i)
char FindFreeMC(char i) {
  t=OPL[i].FreeHd;
  OPL[i].FreeHd=UseFreeL[t];
  return t;
} /* O(1) */
```

Figure 20. Application Tasks for CTDBP (MPT)

DBP, the communication protocol is executed at the kernel level and the application level. Compared with its counterpart shown in Figure 11, this extended version performs exactly the same functionality for a single writer or reader. The only difference is that the extended protocol needs to handle all ports the task may have when activated. Similarly, when a task terminates, the task decrements the use count of the buffer slot that each of its lower-priority input ports points to. If a use count drops to zero, the task further frees this buffer slot by updating the corresponding `FreeHd`. Since `FreeHd` and `UseFreeL[]` are shared by a writer and its lower-priority readers, atomicity of the critical section at termination time must be guaranteed by any correct implementation. Note that a terminating task may have multiple lower-priority input ports. The atomicity is only needed for the series of load and store memory operations for each reader individually, not for the whole process of handling all its lower-priority readers.

Figure 20 shows the constant time search algorithm used in this extension. It is almost exactly the same as its counterpart for CTDBP (SPT) shown in Figure 11, except that it takes in the writer's index as input to return the head of the writer's free list.

The memory consumption of the extended implementation of the CTDBP is shown in Table 6.

variable	char	message
count	5×(NT+SysOP)+4×SysNIP+SysNB	SysNB+SysNOP

Table 6. CTDBP Memory Requirement (MPT)

4.2.2 Temporal Concurrency Control Protocol

In this section, we extend the TCCP presented in Section 4.1.2 to handle tasks with multiple ports. Figure 21 shows the data structures. The size of the shared buffer is computed using Equation 4. Similar to the extension in Section 4.2.1, there are three descriptors to characterize the properties of the application tasks. Due to the simplicity of the nature of the TCCP, the data structures shown in Figure 21 are much simpler than those used for the extended versions of DBPs in Figures 14 and 18.

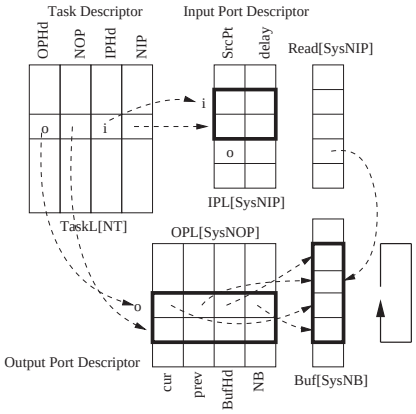

Figure 21. Data Structure for TCCP (MPT)

```
Data Structure
struct TaskEntry {
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT];
```

```
struct OPEntry {
  char cur;
  char prev;
  char BufHd;
  char NB;
} OPL[SysNOP];
```

```
struct IPEntry {
  char SrcPt;
  char delay;
} IPL[SysNIP];
```

```
message WrtInit[SysNOP],Buf[SysNB];
char Read[SysNIP];
```

```
Initialization
OPL[0].cur = OPL[0].prev = OPL[0].BufHd = 0;
Buf[OPL[0].BufHd] = WrtInit[0];
for (i = 1; i < SysNOP; i++) {
  OPL[i].BufHd = OPL[i-1].BufHd + OPL[i-1].NB;
  OPL[i].cur = OPL[i].prev = OPL[i].BufHd;
  Buf[OPL[i].BufHd] = WrtInit[i];
}
```

Figure 22. DS Initialization for TCCP (MPT)

Figures 22 and 23 give the complete TCCP for systems with multi-port tasks. The data structure declaration and the initialization code for the TCCP is shown in Figure 22. Each writer is assigned a continuous segment of `Buf[]`, which is identified by the pair of `BufHd` and `NB` as shown in Figure 21. Similar to the initialization code in Figure 13, `cur`, `prev`, the initial output value are initialized for each writer. The buffer indexing procedure is shown in Figure 23. Comparison with its counterpart for single-port tasks in Figure 13 shows that the extended version performs exactly the same functionality for an individual reader or writer. Since a task may have multiple ports, it needs to process all of them. The constant time `FindFreeMT()` works in the same way as its counterpart as shown in Figure 13, with the difference that it takes as argument the index of the writer for returning the `FreeHd` of the

writer's free list.

```
/* activation time */
/* each writer i */
OPL[i].prev = OPL[i].cur;
OPL[i].cur=FindFreeMT(i);

/* each reader i */
i2 = IPL[i].SrcPt;
if (IPL[i].delay)
  Read[i] = OPL[i2].prev;
else
  Read[i] = OPL[i2].cur;
```

```
/* execution time */
...
/* each writer k */
Buf[OPL[k].cur] = ...
...
/* each reader k */
... = Buf[Read[k]];
...
char FindFreeMT(char idx) {
  return (OPL[idx].cur+1)\
      % OPL[idx].NB;
} /* O(1) */
```

Figure 23. Application Tasks for TCCP (MPT)

The memory consumption of the extended implementation of the TCCP is shown in Table 7.

variable	char	message
count	4×(NT+SysOP)+3×SysNIP	SysNB+SysNOP

Table 7. TCCP Memory Requirement (MPT)

4.2.3 Comparison of (LT/CT)DBP and TCCP

For both the single-port task version and the extended version, though `FindFreeSC()` and `FindFreeMC()` execute in constant time, more operations need to be performed at the kernel level for the writer and lower-priority readers. Furthermore lower-priority readers have to update the shared use free list before their termination and accordingly any correct implementation must guarantee mutual exclusion. In order to gain a constant execution time `FindFree()` algorithm, we are not only paying for more memory but also paying some temporal cost.

Compared with the LTDBP and the CTDBP, the TCCP requires the least amount of memory for auxiliary data structures and it is the simplest. Among these three SR semantics preserving protocols, the TCCP is the only one that can achieve a constant time `FindFree()` without introducing extra data structures and much bookkeeping. However the buffer size based on the temporal concurrency control is highly dependent upon the temporal properties of the writer and reader tasks. Some temporal characteristics may give a buffer size that is much larger than the buffer size that is based on the number of lower-priority reader tasks. The tradeoffs among time, memory, and implementation complexity need to be examined through experiments.

5 OSEK/VDX

From the above discussion, it is clear that protocols preserving the SR semantics need kernel-level support to assign reading and writing buffer indices. This implies that a kernel level implementation is generally required to preserve the synchronous reactive semantics. In this section, we introduce real-time operating system API standards that could be used to implement of the SR semantics preserving protocols presented earlier.

To support portability of real-time application software, RTOS API standards such as OSEK/VDX, POSIX [12], and μITRON [13] have been developed. In this paper, we choose to focus on OSEK/VDX. The OSEK/VDX standards originated from France and Germany and has been widely used in the automotive industry. We now summarize the OSEK OS software architecture, kernel services, and then the OSEK implementation language that is used during system generation.

5.1 OSEK OS Architecture and Functionalities

The OSEK OS architecture is designed to support OS scalability and application software portability. The kernel services are structured into different functionality groups as discussed later in this section.

5.1.1 Software Architecture

Three processing levels are defined in the OSEK OS. From higher to lower priority, they are interrupt level, logical scheduler level, and task level. Tasks are categorized into either basic or extended based on whether they can enter a wait state by calling the `WaitEvent` kernel service. A basic task is not allowed to wait on an event. To support design reuse and ease upgrade, four conformance classes are defined according to the number of active activations of a task, the task type, and the number of tasks per priority level.

To support application portability, minimum requirements are defined for all four conformance classes as shown in Table 8. For example, for BCC1, the minimum requirement specifies single active task activation, eight active tasks, distinct priority assignment for tasks, eight priority levels, one alarm, one application mode, and no event. Any application that meets the minimum requirements is portable to any OSEK-compliant operating systems.

	Basic		Extended	
	BCC1	BCC2	ECC1	ECC2
Multiple Active Task Instances	No	Yes	BT: No ET: No	BT: Yes ET: No
# of Tasks not in Suspend State	8		16 (Any Comb. of BT/ET)	
> 1 Task/Priority	No	Yes	No	Yes
# of Events/Task	-		8	
# of Priority Levels	8		16	
Resources	RES_SCHEDULER		8(including RES_SCHEDULER)	
Internal Resources	2			
Alarm	1			
Application Mode	1			

Table 8. Minimum Requirements for OSEK CC

5.1.2 Kernel Services

In OSEK OS, the kernel functionality includes task management, interrupt management, synchronization, alarm, intra-processor message handling, and error treatment. A task can be activated by either

`ActivateTask` or `ChainTask` and it can only be terminated by itself by calling `TerminateTask`.

An ISR has a statically assigned priority level higher than that of tasks. There are two categories of ISRs specified in the OSEK OS standard. An ISR in category 1 is not allowed to use any kernel services and it cannot be preempted. Termination of an ISR in category 1 does not force any rescheduling. On the other hand, kernel services are allowed in an ISR in category 2. At the end of its execution, rescheduling will be performed if there is no other pending ISRs.

Synchronization can be achieved by using events and semaphores. The kernel primitive `WaitEvent` is only accessible to extended tasks. An event is owned by an extended task and it can be set by either a basic task, an extended task, or even an ISR in category 2. An event is non-consumable and therefore it needs to be cleared by its owner after being used. Another synchronization mechanism used by tasks is semaphores. Both event and semaphore are a blocking mechanism.

Alarms are managed in a layered manner. On the OSEK OS kernel side, counters are measured in ticks and at least one counter is generated from a hardware or software timer. On the application side, primitives managing alarms are provided. An alarm can be associated with only one counter, but a counter can be used as a reference for more than one alarm. An alarm can be used to activate a task, set an event, or call an alarm callback routine. OSEK supports relative and absolute alarms and an alarm can be either single or cyclic.

Messages are used as a means for intra-processor communication. Similarly, minimum functionality is defined in [14] for different communication conformance classes.

The hook routine mechanism is used for error handling, tracing, and debugging purposes. This mechanism allows application specific functionalities to be processed internally by the OSEK OS. As part of the OS, a hook routine has a priority that is higher than all tasks and it cannot be preempted by ISRs in category 2. Table 9 summarizes kernel services defined in the OSEK OS standard.

Task Model	Basic task; Extended task
Synchronization	Event; Semaphore
Semaphore Sync Protocol	Priority Ceiling Protocol (Highest Locker Protocol)
Inter-task Comm Mechanism	Global variable; Message; Message filtering and notification
Task Management	Activation, termination, chaining, and state reference
Scheduling	Non-/full/mixed preemptive; on the same level First Come First Served
Multiple Activation	BCC2 tasks and basic tasks in ECC2
Memory Management	No virtual memory (MMU); No dynamic allocation
Stack Sharing	Yes for BCC and No for ECC
Interrupt Handling	ISR category 1 and 2; Nesting allowed
Time Management	Alarm callback routine; Counter/alarm (relative/absolute; single/cyclic)
Error Management Tracing and Debugging	User-defined hook routine; Error code; Application error((de)centralized), fatal error(centralized shutdown)

Table 9. Summary of the OSEK OS Standard

5.2 OSEK Implementation Language

To support modular configuration for system generation of an application, the OSEK Implementation Language (OIL) has been designed. In this section, we first show the OSEK application development flow and then get into the detailed contents of OIL configuration files.

5.2.1 Application Development Process

Figure 24 illustrates a sample OSEK develop process for applications. An OIL configuration file can be prepared manually or generated automatically. Then the OIL files are fed to System Generator (SG), which automatically configures a kernel through choosing the required modules and customizing the data structure attributes based on the configuration file. Finally the application source code directly from users, the selected module files from the OSEK OS kernel library, and the additional application file produced by SG are compiled and linked together to produce an executable file for the application.

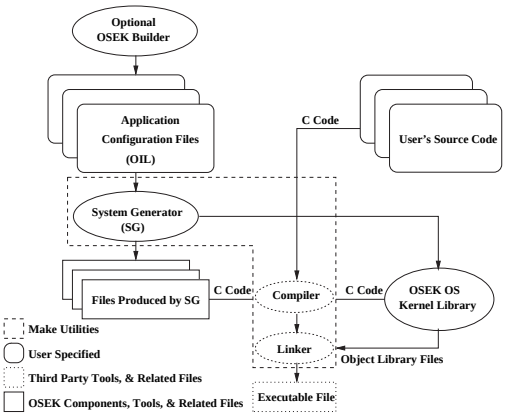

Figure 24. Application Development Process

5.2.2 What Is inside an OIL Configuration File

OIL is a mechanism used to configure an OSEK application inside a particular CPU. The OIL description of an OSEK application consists of a set of OIL objects that are characterized by a set of attributes and references. Attributes and references can be either standard or optional (application specific). Refer to Table 10 for all OSEK OIL objects and their properties.

An OIL configuration is composed of two parts: implementation definition and application definition. The former defines all standard and application-specific attributes and their properties for a particular OSEK implementation while the latter defines the set of objects and their corresponding attribute values for an OSEK

Object	Mandatory	Standard Attribute	Std Reference
CPU	yes	-	-
OS	yes (= 1)	STATUS; USERESSCHEDULE; USEGETSERVICEID; Hooks; USEPARAMETERACCESS	-
APPMODE	yes (≥ 1)	-	-
TASK	yes (≥ 1)	PRIORITY; SCHEDULE; ACTIVATION; AUTOSTART	MESSAGE; EVENT; RESOURCE
COUNTER	no	MAXALLOWEDVALUE; TICKSPERBASE; MINCYCLE	-
RESOURCE	no	RESOURCEPROPERTY	-
EVENT	no	MASK	-
ISR	no	CATEGORY	MESSAGE;RESOURCE
MESSAGE	no	NOTIFICATION; etc.	-
NWMESSAGE	no	SIZEINBITS; etc.	IPDU
COM	no (= 1)	COMTIMEBASE; etc.	-
IPDU	no	SIZEINBITS; etc.	-
NM	no (= 1)	-	-

Table 10. OIL Objects and Their Properties

```
Compute TSize
char TSize = 0;
for (i = 0; i < NT; i++) {
  TSize += \
        LCMR/TaskL[i].rate;
}

Declaration
struct TickEntry {
  char DispHd;
  char size;
} TickL[LCMR];

char tick;
char DTab[TSize];
```

```
Init w/o Phase Shift
tick = -1;
i1 = 0;
for (j=0; j<LCMR; j++) {
  TickL[j].DispHd = -1;
  TickL[j].size = 0;
  for (i=0; i<NT; i++) {
    if(j%TaskL[i].rate==0){
      i2=TickL[j].size+i1;
      DTab[i2] = i;
      TickL[j].size++;
  } }
  if (TickL[j].size!=0) {
    TickL[j].DispHd = i1;
    i1 += TickL[j].size;
} }
```

Figure 26. Task Dispatcher

application. All attributes used in an application definition must be defined in the corresponding implementation definition.

6 OSEK Implementations of SR Semantics Preserving Protocols

Based on the previous two sections, we present OSEK implementations of the SR semantics preserving protocols under BCC1. We further assume that all tasks are periodic. In particular we are interested in OSEK implementations that are portable. Any application implementation that meets the minimum requirements is portable to any OSEK-compliant RTOS. From Table 8, we know that there is only one alarm available for use. In the rest of this section, we first present how to implement an application in OSEK with using a single alarm and then show the development of the OIL configuration file.

6.1 Design of Application Implementation

In BCC1 and BCC2, events are not available and the alarm mechanism is the only way to activate periodic tasks. Since the minimum requirement specifies only one alarm, we use this alarm to periodically activate a task called `dispatcher` that activates other application tasks at their respective proper activation time. On behalf of the kernel, the task `dispatcher` defines the proper buffer indices for writers and readers upon their activation. This implies that the `dispatcher` executes at the kernel level. To handle all different periods of application tasks, the task `dispatcher` executes at a

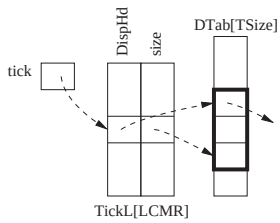

Figure 25. Task Dispatcher Data Structure

rate of `GCDR`, which denotes the Greatest Common Divisor (GCD) of the Rates of all application tasks. The alarm is statically configured to be a cyclic alarm with a period of `GCDR`.

The data structures for the task `dispatcher` are shown in Figure 25 and the corresponding declaration is shown in Figure 26. Array `TickL` has a dimension of `LCMR`, standing for the Least Common Multiple of the Rates of application tasks. Each entry of `TickL` has two fields: `DispHd` and `size`. `DispHd` points to the first task on the `DisTab[]` and `size` indicates the number of tasks that need to be activated at this `tick`. Array `DTab[]` records the tasks that need to be activated from `tick = 0` to `tick = LCMR-1`. The size of array `DTab[]`, `TSize`, is calculated as in Figure 26. The entries of `DTab[]` are used to index the tasks in the task descriptor presented earlier.

The right column in Figure 26 is the initialization of the data structures used for task `dispatcher`. At each tick `j`, all tasks that need to be activated are recorded into the dispatch table and the fields `DispHd` and `size` are specified accordingly. The detailed definition of the task `dispatcher` will be discussed later in this section.

Recall that the data structures of the communication protocols discussed in Section 4 need to be initialized to obtain correct execution semantics. These can be done in an OSEK task called `init` as shown in Figure 27 or alternatively these can be done in the main function of the application during the system starts up. Task `init` executes once when the system starts up. In Section 4, we assume only data structures of the protocols themselves need to be initialized and system information such as the relative priority is all known. In practice, only `rate`, `pri`, and `delay` are given. To speed up normal execution, some static information such as the `IsHPR` in the task descriptor needs to be computed at system startup and stored for later use. Besides initializing data structures, task `init` also sets the cyclic alarm, `dispAlarm`, associated with the task

dispatcher. At the end, init calls the OSEK kernel primitive TerminateTask to terminate itself.

```
TASK (init) {
  ··· /* init implementation specific auxiliary DS */
  ··· /* init DS required by protocol */
  ··· /* init dispatcher as in Fig 26 */
  SetRelAlarm(dispAlarm, 0, GCDR);
  TerminateTask();
}
```

Figure 27. General Structure of Task Init

In summary, for a portable OSEK implementation under BCC1, there is dispatcher, init, and one or more application tasks. All OSEK tasks need to be declared as shown in Figure 28 before being used.

```
DeclareTask(AppTask_i);          DeclareTask(init);
···                              DeclareTask(dispatcher);
```

Figure 28. OSEK Task Declaration

6.2 Task Implementation in C Code

In this section, we present complete OSEK implementations of the SR semantics preserving communication protocols. As in Section 4, we first present implementations for applications only with single-port tasks and then show the corresponding extended versions. For each implementation, we show the required data structures, the corresponding declaration, the definitions for dispatcher, application tasks, and init.

6.2.1 Implementation for Single-Port Tasks

In Section 4.1, three SR semantics preserving protocols are presented: LTDBP, CTDBP, and TCCP for applications with single-port tasks. In this section, we first present common data structures and the definition of the task dispathcer for an OSEK BCC1 implementation of applications using these protocols. Figure 29 shows the common data structure declaration for systems with single-port tasks.

```
#DEFINE NR X          #DEFINE NT X          #DEFINE TSize X
#DEFINE LCMR X        #DEFINE NB X
#DEFINE GCDR X        #DEFINE WrtInit X
message Buf[NB];      char cur, prev;       char Read[NR];
```

Figure 29. Common DS Declaration (SPT)

Figure 30 shows the general structure of the task dispatcher used for applications with single-port tasks. Each time its alarm goes off, the dispatcher's state changes to ready and gets executed. Counter tick is incremented and the modulus operation is taken with respect to LCMR. The remainder is reassigned to tick. Then the value of the field DispHd of the current TickL[tick] entry is checked. If it is "-1", no task needs to be activated at this tick. Otherwise, all the tasks specified by DispHd and size in DTab[] need to be handled by assigning the reading/writing index activating this task. Finally the task dispatcher calls the OSEK API TerminateTask for self-termination.

```
TASK(dispatcher) {
  tick = (tick+1) % LCMR;
  if (TickL[tick].DispHd != -1) {
    for (k = 0; k < TickL[tick].size; k++) {
      idx = DTab[k+TickL[tick].DispHd];
      if (idx == (NT-1)) { /* writer */
        ··· /* kernel-level writer code */
      }
      else { /* reader */
        ··· /* kernel-level reader code */
      }
      ActivateTask(idx);
  } }
  TerminateTask();
}
```

Figure 30. Task Dispatcher for SPT

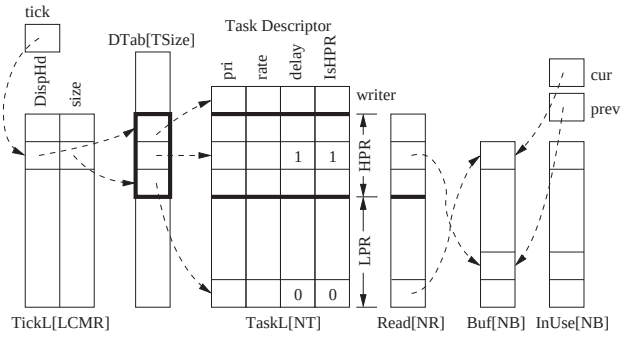

Figure 31. LTDBP Implementation DS (SPT)

Implementation of LTDBP Figure 31 illustrates the data structures used in our implementation. It is a combination of the data structures used for the LTDBP in Figure 7 and the data structures used for the task dispatcher in Figure 25. Comparing with the task descriptor in Figure 7, pri and rate are added into the task descriptor for initialization of the field IsHPR and the data structures of task dispatcher, respectively. Figures 32 and 26 show the corresponding data structure definition.

```
struct TaskEntry {                    message Buf[NB];
  char pri;                           char cur, prev;
  char rate;                          char Read[NR];
  char delay;                         char InUse[NB];
  char IsHPR;
} TaskL[NT] = {{X,X,X,X}, ···};
```

Figure 32. LTDBP DS Declaration (SPT)

The implementation of the task dispatcher has been shown in Figure 30. The task dispatcher performs kernel-level operations in Figure 8 based on whether it is writer or reader. Specifically it calls FindFreeSL() to find a safe buffer slot for the writer to write data into during execution. For a reader, the dispatcher defines the buffer slot that the reader would read from during its execution.

Figure 33 shows the OSEK implementation for ap-

```
TASK (writer) {
  ...
  Buf[cur] = ···
  ...
  TerminateTask();
}
```

```
TASK (reader_j) {
  ...
  ··· = Buf[Read[j]];
  ...
  if (TaskL[j].IsHPR == 0)
    Read[j] = -1;
  TerminateTask();
}
```

Figure 33. Writer/Reader for the LTDBP (SPT)

plication tasks, where only the portion relevant to the protocol implementation is shown. Specifically, the writer writes data into the buffer slot that has been defined at its activation time by the task `dispatcher` at the kernel level. Similarly the reader gets data from the buffer slot assigned by the `dispatcher` and flags its completion on termination if it is a lower-priority reader. Note that mutual exclusive access to the shared `Read[]` is guaranteed by the fact that single memory operation is atomic. At the end of their execution, application tasks terminate themselves by calling the OSEK API `TerminateTask`.

```
TASK (init) {
  /* init TaskL[].IsHPR */
  for (j=1; j<(NT-1); j++) {
    if (TaskL[j].pri \
          > TaskL[NT-1].pri)
      TaskL[j].IsHPR = 1;
    else
      TaskL[j].IsHPR = 0;
  }
  ···
}
```

Figure 34. Part of Init for the LTDBP (SPT)

The general structure of task `init` has been shown in Figure 27. The data structures for the LTDBP are initialized as shown in Figures 8 and the initialization of the `IsHPR` is shown as in Figure 34.

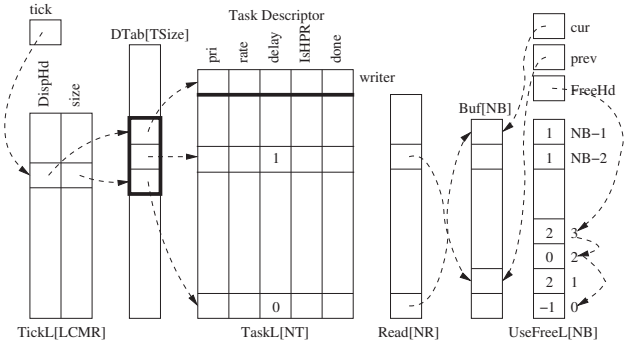

Figure 35. CTDBP Implementation DS (SPT)

Implementation of CTDBP In this section, we present an OSEK application implementation when using the CTDBP shown in Figure 11. Figure 35 illustrates the data structures used in the implementation. Similar to the implementation for applications with LTDBP, besides fields `pri` and `rate`, a third field called `done` is further introduced for the purpose of guaranteeing when the critical section at the end of lower-priority readers needs to be executed. Figures 36 and 26 show the corresponding declaration.

```
struct TaskEntry {
  char pri;
  char rate;
  char delay;
  char IsHPR;
  char done;
} TaskL[NT] = {{X,X,X,X,0},··· };
```

```
char FreeHd;
char UseFreeL[NB];
```

Figure 36. DS Declaration for CTDBP (SPT)

The task dispatcher required in this OSEK application implementation has the same functionality as shown in Figure 30: at activation time to find and assign a safe buffer slot for a writer and assign the buffer index a reader should read from during execution. The corresponding kernel-level application code from Figure 11 is executed for the CTDBP.

```
TASK (writer) {
  ...
  Buf[cur] = ···
  ...
  TerminateTask();
}
```

```
void PostTaskHook(void) {
  GetTaskID(id);
  if (TaskL[id].done) {
    ··· /* termination code
                in Fig 11*/
} }
```

```
TASK (reader_j) {
  TaskL[j].done = 0;
  ...
  ··· = Buf[Read[j]];
  ...
  TaskL[j].done = 1;
  /* atomic termination
       by PostTaskHook() */
  TerminateTask();
}
```

Figure 37. Writer/Reader and `PostTaskHook` for CTDBP (SPT)

Figure 37 shows an OSEK implementation of application tasks, in which only the portion relevant to the application level execution of the CTDBP is shown. From the discussion in Section 4.1.1, we know that the constant execution time of the search algorithm `FindFreeSC()` is obtained through using a use free list. Lower-priority tasks need to update this list atomically at termination time. Our implementation must guarantee this requirement. Semaphores are a way to provide mutual exclusion, but they are not an option here since the DBP is a wait-free protocol and use of semaphore will defeat the whole purpose.

In this paper, we propose to use the hook mechanism provided in the OSEK OS standards to achieve atomicity for a critical section. Specifically, we use the `PostTaskHook` to gain a kernel-level execution for this critical section on lower-priority tasks' termination. Since the `PostTaskHook` routine executes at each context switch of all tasks in the system, we introduce a flag called `done` in the task descriptor as described earlier in this section. Flag `done` serves two purposes. First, it indicates for which tasks the `PostTaskHook` needs to be executed. This is achieved by setting `done` to be false (0) and never turn it to true for tasks that do not need the functionality in the `PostTaskHook`.

The second purpose is to assure that operations in the `PostTaskHook` are only executed on task termination time instead of during each context switch. This can be achieved by setting the task's `done` flag to be false at the beginning of the body of the task and turn the flag to true (1) right before it finishes.

In the definition of the `PostTaskHook`, it first obtains the identifier of the task that triggers the execution of the `PostTaskHook` routine by calling the OSEK API `GetTaskID`. And then it checks whether or not the `done` flag of this task is true. If the task terminates, then the required application-level functionality of the communication protocol is performed. As part of the operating system, the hook mechanism guarantees the atomicity of critical sections naturally.

Alternatively, APIs `SuspendOSInterrupts` and `ResumeOSInterrupts` could be used to gain exclusive access to the shared use free list.

The `init` task required in this OSEK application implementation performs the same functionalities as its counterpart in Figures 27 and 34. The only difference is that the initialization code for the protocol is from Figure 11 for the CTDBP.

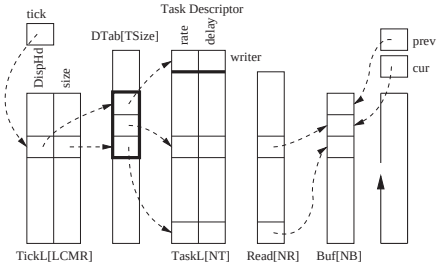

Figure 38. TCCP Implementation DS (SPT)

Implementation of TCCP Figure 38 shows the data structures for implementing applications using the TCCP. It is the combination of data structures in Figures 25 and 12 with slight modifications. In the task descriptor, only field `rate` is added, which leads to a two-field entry for each task on `TaskL`. Comparison with Figures 31 and 35 shows that the data structures for the TCCP implementation is the simplest one since there is no bookkeeping data structure used for the `FindFreeST()`.

The task `dispatcher` required in the OSEK application implementation for TCCP has the same functionality as shown in Figure 30. The corresponding kernel-level application code from Figure 13 is executed for the TCCP.

The definition of application tasks shares the same structures with its counterpart shown in Figure 33 and it is even simpler because there is no bookkeeping termination code for reader tasks.

The `init` task required in this OSEK application implementation share a similar but simpler structure compared with the general `init` in Figures 27 since the field `IsHPR` is not needed. It performs the operations shown in Figure 13 to initialize the data structures for the TCCP itself.

6.2.2 Implementation for Multi-Port Tasks

In this section, we extend the implementations in Section 6.2.1 to deal with systems with multiple-port tasks. The declaration for common data structures used in implementations is shown in Figure 39.

```
#DEFINE NT X          #DEFINE SysNIP X     #DEFINE TSize X
#DEFINE LCMR X        #DEFINE SysNOP X     #DEFINE SysNB X
#DEFINE GCDR X
message Buf[SysNB];                        char Read[SysNIP];
message WrtInit[SysNOP] = {X, ···};
```

Figure 39. Common DS Declaration (MPT)

```
TASK(dispatcher){
  tick = (tick+1) % LCMR;
  if (TickL[tick].DispHd != -1) {
    for (k = 0; k < TickL[tick].size; k++) {
      idx = DTab[k+TickL[j].DispHd];
      for (i = 0; i < TaskL[idx].NOP; i++) {
        idx2 = TaskL[idx].OPHd + i;
        ··· /* kernel level writer code */
      }
      for (i = 0; i < TaskL[idx].NIP; i++) {
        idx2 = TaskL[idx].IPHd + i;
        ··· /* kernel level reader code */
      }
      ActivateTask(idx);
  } }
  TerminateTask();
}
```

Figure 40. Dispatcher for MPT

Figure 40 shows the definition of the `dispatcher` used for systems with multiple-port tasks. Similar to the dispatcher presented in Figure 30, it activates application tasks according to a dispatch table precomputed in the task `init`. The only difference is that a task now may have multiple input/output ports. Before activating a task, the dispatcher processes all its ports by performing kernel level execution for each port, which can be either a reader or a writer. Then the `dispatcher` activates the task using the API `ActivateTask`. At the end of the definition, the `dispatcher` calls `TerminateTask` for termination.

Implementation of LTDBP Figure 41 shows the data structures needed for our OSEK implementation of applications with multi-port tasks using the LTDBP. It combines the data structures in Figures 14 and 25 with slight modification. Besides fields `rate` and `pri`, field `owner` is added into both the input port and output port descriptors. Figures 42, 39, and 26 together show the complete corresponding data structure declaration. The task `dispatcher` shares the same structure as shown in Figure 40 and the corresponding kernel-

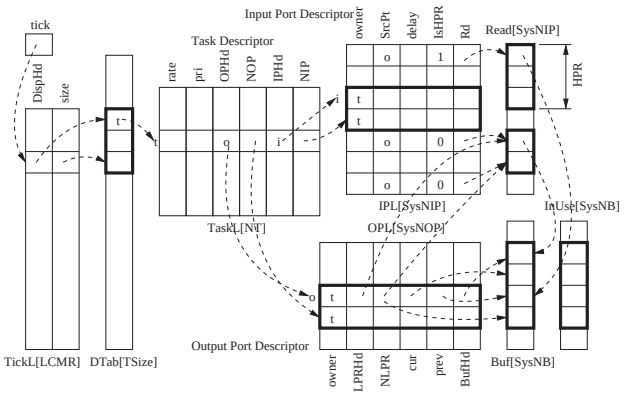

Figure 41. LTDBP Implementation DS (MPT)

level application code is from Figure 16.

```
struct TaskEntry{
  char rate;
  char pri;
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT] = {
  {X,X,X,X,X,X},
   ···};
```

```
struct OPEntry {
  char owner;
  char NLPR;
  char LPRHd;
  char cur;
  char prev;
  char BufHd;
} OPL[SysNOP] = {
  {X,X,X,X,X,X},
   ···};
```

```
struct IPEntry {
  char owner;
  char SrcPt;
  char delay;
  char IsHPR;
  char Rd;
} IPL[SysNIP] = {
  {X,X,X,X,X},
   ···};
char InUse[SysNB];
```

Figure 42. DS Declaration for LTDBP (MPT)

```
TASK (AppTask_i) {
  ···
  /* each writer k */
  Buf[OPL[k].cur] = ···
  ···
  /* each reader k */
  ···= Buf[Read[IPL[k].Rd]];
  ···
```

```
  /* termination time */
  /* each LP reader */
  if (IPL[k].IsHPR==0) {
    Read[IPL[k].Rd] = -1;
  }
  TerminateTask();
}
```

Figure 43. OSEK Implementation of Application Tasks for LTDBP (MPT)

Figure 43 shows the OSEK implementation of the communication protocol implemented at the application level. Comparison with its counterpart in Figure 33 shows that they perform the same functionality for a single reader or writer. But the application task shown in Figure 43 needs to process all ports it may have. When a task has multiple lower-priority readers, it need to flag the completion for all these readers.

Note that our implementation guarantees that the operations in the termination code are executed atomically for a single reader. Making the whole process for all these lower-priority readers' termination code atomic is not required for a correct execution semantics and even not desirable because that leads to a long critical section.

The `init` task required for this OSEK application implementation has the same structure as the general `init` in Figures 27. Specifically, the initialization code for the LTDBP is from Figure 15 and the auxiliary field `IsHPR` is initialized as shown in Figure 44.

```
TASK (init) {
  /* init IPL[].IsHPR */
  for (j=1; j<(SysNIP-1); j++) {
    srcPt = OPL[IPL[j].SrcPt;
    if(TaskL[IPL[j].owner].pri>\
        TaskL[srcPt].owner].pri)
      IPL[j].IsHPR = 1;
```

```
    else
      IPL[j].IsHPR=0;
  }
  ···
}
```

Figure 44. Part of Init for the LTDBP (MPT)

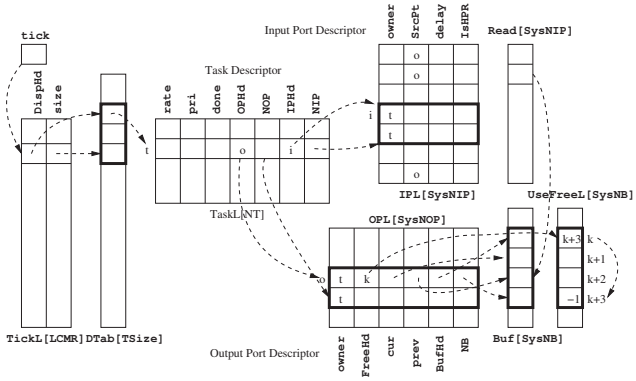

Figure 45. CTDBP Implementation DS (MPT)

Implementation of CTDBP Figure 45 shows the data structures used for our OSEK implementation for systems with multiple-port tasks using the CTDBP. It combines the data structures for the task dispatcher in Figure 25 and the corresponding data structures for the CTDBP in Figure 18. Besides `rate` and `owner`, a new field called `done` is added to the task descriptor for the same purpose as discussed for its counterpart in Figure 35. The corresponding data structure declaration is shown in Figures 46, 39, and 26.

```
struct TaskEntry {
  char rate;
  char pri;
  char done;
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT] = {
  {X,X,0,X,X,X,X},
  ···};
```

```
struct OPEntry {
  char owner;
  char FreeHd;
  char cur;
  char prev;
  char BufHd;
  char NB;
} OPL[SysNOP] = {
  {X,0,0,0,0,X},
  ···};
```

```
struct IPEntry {
  char owner;
  char SrcPt;
  char delay;
  char IsHPR;
} IPL[SysNIP] = {
  {X,X,X,X},
  ···
};
char UseFreeL[SysNB];
```

Figure 46. DS Declaration for CTDBP (MPT)

The task dispatcher shares the same structure as shown in Figure 40 and the corresponding kernel-level application code is from Figure 20.

Figure 47 shows the implementation of application tasks. It is actually a combined version of single-port writer and reader tasks shown in Figure 37. The same hook mechanism is used to guarantee the atomicity of the termination code for lower-priority readers and the `PostTaskHook` is similarly defined. For each terminating low-priority reader, the two versions of the

```
TASK (AppTask_i) {
  TaskL[i].done = false;
  ...
  Buf[OPL[k].cur] = ...
  ...
  ... = Buf[Read[k]];
  ...
  TaskL[i].done = true;
  /* atomic hook code */
  TerminateTask();
}
```

```
void PostTaskHook(void) {
  char id, j, k, nip, t1, t2;
  GetTaskID(id);
  if (TaskL[id].done) {
    nip = TaskL[id].NIP;
    for (j=0; j<nip; j++) {
      k = j + TaskL[id].IPHd;
      ... /* CS in Fig 20 */
    }
  }
}
```

Figure 47. OSEK Implementation of Application Task for CTDBP (MPT)

`PostTaskHook` perform the same operations on their use free lists. The only difference is that the version shown in Figure 47 needs to process all lower-priority input ports that a terminating task may have.

The `init` task required for this OSEK application implementation performs the same functionalities as its counterpart in Figures 27 and 44. The only difference is that the initialization code for the CTDBP is from Figure 19.

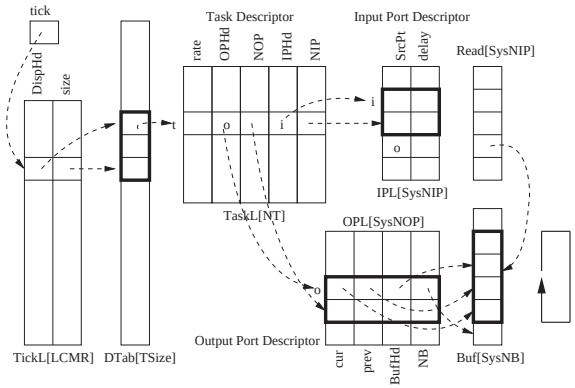

Figure 48. TCCP Implementation DS (MPT)

Implementation of TCCP Figure 48 shows the data structures used for the OSEK implementation for systems with multiple-port tasks using the TCCP. It is the combination of the data structures in Figures 25 and 21 with field `rate` added to the task descriptor. The complete corresponding declaration is shown in Figure 49, 39, and 26.

```
struct TaskEntry {
  char rate;
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT] = {
  {X,X,X,X,X}, ...};
```

```
struct OPEntry {
  char cur;
  char prev;
  char BufHd;
  char NB;
}OPL[SysNOP] = {
  {0, 0, 0, X},
  ...}
```

```
struct IPEntry {
  char SrcPt;
  char delay;
}IPL[SysNIP] = {
  {X, X},
  ...
};
```

Figure 49. DS Declaration for TCCP (MPT)

The task `dispatcher` performs the same functionality as shown in Figure 40 and the corresponding kernel-level application code is from Figure 23.

The definition of application tasks shares the same structure with its counterpart shown in Figure 43 with the exception that it does not have bookkeeping termination code in the protocol.

The `init` task required for this OSEK application implementation has a similar but simpler structure compared with the general `init` in Figure 34 since the field `IsHPR` is not needed. The initialization code for the TCCP is from Figure 22.

6.2.3 OSEK Implementation Comparison

In this section, we compare the implementations presented earlier. Table 11 shows the memory requirements for the application implementations with different versions of SR semantics preserving protocols. For applications with single-port tasks, the CTDBP requires the most amount of auxiliary data structures and the TCCP requires the least amount of auxiliary data structures. LTDBP and CTDBP need the same amount of buffer slots. The buffer sizes for DBP and TCCP are not comparable since they are based on completely different buffer sizing mechanisms. The same trend holds for the counterpart implementations for multiple-port tasks except that the implementations for the LTDBP and the CTDBP require about the same amount of chars.

protocol	char	message
SLTDBP	5×NR+2×LCMR+TSize+NB+7	NB
SCTDBP	6×NR+2×LCMR+TSize+NB+9	NB
STCCP	3×NR+2×LCMR+TSize+5	NB_T
MLTDBP	6×(NT+SysNOP+SysNIP)+ 2×LCMR+SysNB+TSize+1	SysNB+ SysNOP
MCTDBP	7×NT+6×SysNOP+5×SysNIP+ 2×LCMR+SysNB+TSize+1	SysNB+ SysNOP
MTCCP	5×NT+4×SysNOP+3×SysNIP+ 2×LCMR+TSize+1	$SysNB_T$+ SysNOP

Table 11. Memory Requirement Comparison

Since the hook mechanism is mainly designed for debugging and error treatment purposes, using the `PostTaskHook` to gain atomicity for user-defined functionality brings overheads. First of all, the `PostTaskHook` executes during each context switch and the number of context switches may be big. Secondly, the use of the `PostTaskHook` may increase the size of the footprint of the application.

As far as the implementation complexity, for both versions, the implementation for applications using TCCP is the least complex due to its nature. The implementation for applications using the CTDBP is the most complex because of the sophisticated protocol definition, the supporting data structures, and the necessity to guarantee the atomicity of the termination code for tasks with lower-priority readers.

6.3 OIL Configuration File

OIL files are a means to statically configure OSEK application implementations. In this section, an OIL

configuration file for implementations of the communication protocols is presented.

```
OIL_VERSION = "2.5";
/* Implementation Def */
IMPLEMENTATION myOSEKOS {
  ...
}; // End of myOSEKOS
/* Application Def */
CPU myCPU { // container
  /* OS Object */
  OS myOS {
    STATUS = STANDARD;
    STARTUPHOOK = FALSE;
    ERRORHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = TRUE;
    USEGETSERVICEID = FALSE;
    USERESSCHEDULER = FALSE;
  };
  /* Task Object */
  TASK AppTask_j {
    PRIORITY = X_j;
    SCHEDULE = FULL;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
  };
  ...
  TASK dispatcher {
    PRIORITY = X_d;
    SCHEDULE = NON;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
  };

  TASK init {
     PRIORITY = X_i;
     SCHEDULE = NON;
     ACTIVATION = 1;
     AUTOSTART = TRUE {
       APPMODE = AppMode0;
     };
  };
  /* Alarm Object */
  ALARM dispAlarm {
    COUNTER = SysTimer;
    ACTION = ACTIVATETASK{
      TASK = dispatcher;
    };
    AUTOSTART = TRUE {
      ALARMTIME = 0;
      CYCLETIME = GCDR;
      APPMODE = AppMode0;
    };
  };
  /* Counter Object */
  COUNTER SysTimer {
    MINCYCLE = x;
    MAXALLOWEDVALUE = x;
    TICKSPERBASE = x;
  };
  /* Appl Mode Object */
  APPMODE AppMode0 {
    VALUE = AUTO;
  };
}; // End of myCPU
```

Figure 50. OIL Configuration File

Figure 50 shows the basic structure of an OIL configuration file. In this file, the version number of OIL is first specified, which is 2.5 in our example. Then the implementation definition section follows. This definition usually comes from RTOS vendors and we do not need to modify it for our development.

The next section is the application definition, which is application-specific. Inside the container `CPU`, objects are statically specified. In our implementation, we have application tasks (`AppTask_j`). The application tasks' priorities are statically specified by designers. Under a preemptive scheduling, we set the `SCHEDULE` attribute as `FULL`, which indicates a fully preemptive scheduling policy. Under the assumption that the deadlines of application tasks are not greater than their respective periods, the `ACTIVATION` attribute is set as one (as required by BCC1). Application tasks are periodic and will be activated by the task `dispatcher`, therefore the attribute `AUTOSTART` is set as `FALSE`. For task `init`, the configuration is specified similarly. The `AUTOSTART` attribute is turned on for task `init` and an application mode is assigned to the `APPMODE` attribute. The properties of the specified application mode need to be defined in the `APPMODE` object. The `dispatcher` activates application tasks and performs part of the communication protocol operations on behalf of the kernel. Therefore its priority should be higher than those of all application tasks. Since it executes like inside the kernel, its `SCHEDULE` attribute is set to be `NON`, which represents a non-preemptive scheduling policy. The `dispatcher` is activated by an alarm, so the `AUTOSTART` attribute is set to be `FALSE`. There is no pending activations for the `dispatcher`. Because alarm `dispAlarm` is used by the task `dispatcher`, an alarm object is specified accordingly. The alarm is associated with a counter, which is another object and statically defined in the OIL file. The alarm is configured to activate task `dispatcher` through setting its attribute `ACTION` as `ACTIVATETASK`. Finally the alarm's `AUTOSTART` attribute is set as `TRUE` and explicitly associating the task `dispatcher` that needs to be activated. The period of `dispAlarm` is assigned as `GCDR`.

When the constant time `FindFreeCT()` is used, the atomicity of the termination code that updates the shared use free list is guaranteed by the `PostTaskHook` mechanism through setting the corresponding attribute `POSTTASKHOOK` in the OS object as shown in Figure 50.

7 Conclusions and Further Work

In this paper, we presented portable OSEK implementations for the synchronous reactive semantics preserving communication protocols. We showed detailed data structures and imperative code for two versions of the dynamic buffering protocol: a linear time and a constant time `FindFree()` for single-port and multiple-port tasks. We also presented OSEK implementations for applications using the TCCP. To meet the minimum requirements of BCC1 for portability, only one alarm is used in the implementations to periodically activate a task `dispatcher` that activates application tasks at their proper activation time. When a constant time `FindFree()` is used in the DBP, the atomicity of the critical section at the end of application tasks with lower-priority readers is obtained by using the `PostTaskHook` mechanism.

Comparison of different versions of the implementation shows that the TCCP implementation uses the least amount of auxiliary data structures and it has the lowest implementation complexity. On the other hand, the CTDBP and the LTDBP use similar amounts of auxiliary data structures. The CTDBP has the highest implementation complexity due to the management of the use free list to support a constant time search algorithm. In addition, the implementations of the CTDBP may have a bigger footprint than the LTDBP because modules for the hook mechanism exist in the final image of the CTDBP.

As ongoing work, we are conducting experiments with different versions of the implementations of the

presented protocols to study and verify the temporal, spatial, and implementation complexity tradeoffs.

References

[1] J. Chen and A. Burns, "A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems," Tech. Rep. YCS 286, Department of Computer Science, University of York, January 1997.

[2] N. Scaife and P. Caspi, "Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems," in *6th Euromicro Conference on Real-Time Systems (ECRTS'04)*, July 2004.

[3] J. Chen and A. Burns, "A fully asynchronous reader/write mechanism for multiprocessor real-time systems," Tech. Rep. YCS 288, Department of Computer Science, University of York, May 1997.

[4] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," *Proceedings of the 6th ACM EMSOFT conference*, October 2006.

[5] J. Chen and A. Burns, "Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties," in *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, (Washington, DC, USA), p. 236, IEEE Computer Society, 1999.

[6] H. Kopetz and J. Reisinger, "The non-blocking write protocol nbw: A solution to a real-time synchronization problem," in *Proceedings of the 14th IEEE Real-Time Systems Symposium*, December 1993.

[7] M. Baleani, A. Ferrari, L. Mangeruca, and A. S. Vincentelli, "Efficient embedded software design with synchronous models," in *Proceedings of the 5th ACM EMSOFT conference*, 2005.

[8] OSEK, "OSEK OS, version 2.2.3." available at `http://www.osek-vdx.org`.

[9] OSEK, "OSEK implementation language (OIL), version 2.5." available at `http://www.osek-vdx.org`.

[10] Mathworks, *The Mathworks Real-Time Workshop User's Guide: Modeling, Simulation, Implementation*. web page: http://www.mathworks.com.

[11] S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi, "Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or edf schedulers," *Proceedings of the 5th ACM EMSOFT conference*, 2005.

[12] POSIX, "POSIX standard." available at `http://www.posix.com`.

[13] microITRON, "microITRON standard, version 4.0." available at `www.sakamura-lab.org/TRON/ITRON/DOC/iim00/microITRON4.pdf`.

[14] OSEK, "OSEK COM, version 3.0.3." available at `http://www.osek-vdx.org`.