Spatial and Temporal Cost Analysis on OSEK Implementations of Synchronous Reactive Semantics Preserving Communication Protocols

Guoqiang Wang
Marco Di Natale
Alberto L. Sangiovanni-Vincentelli

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-149
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-149.html

December 6, 2008

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.

Spatial and Temporal Cost Analysis on OSEK Implementations of Synchronous Reactive Semantics Preserving Communication Protocols*

Guoqiang Wang[1] geraldw@eecs.berkeley.edu
Marco Di Natale[2] marco@sssup.it
Alberto Sangiovanni-Vincentelli[1] alberto@eecs.berkeley.edu
[1] University of California at Berkeley, CA, USA
[2] Scuola Superiore S. Anna, Pisa, Italy

10/17/2007

Abstract

Synchronous Reactive semantics preserving communication buffer sizing mechanisms and buffer indexing protocols are presented. 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. Memory requirements are compared quantitatively for different versions of implementations of the SR semantics preserving protocols. Furthermore run time characteristics for different versions of implementations are measured with the use of the PICos18.

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

*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.

blocking. At the end of the reading operation, it performs a validity check on the data. If realizing there 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 the 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. `NR` 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 channel. 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. For brevity, OSEK is commonly used. 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 already 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 we implemented and analyzed inter-task communication protocols with kernel-level support to preserve the SR semantics. We 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 quantitatively analyzed the temporal and spatial complexities of different versions of implementations.

The paper is the continuation of [11]. It 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 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. Performance analysis of the implementations under a specific OSEK- compliant RTOS called PICos18 is given in Section 7. The paper is concluded with Section 8.

2 Definitions and SR 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.

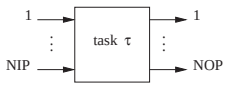

Figure 1. A Task with Multiple Input and Output Ports

Task τ_i is characterized by a set of parameters: priority π_i, period (periodic or sporadic task) τ_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 general, 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 task, a writer task, or both. A reader and a writer correspond to an input port and an output port, respectively.

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 produced by the j^{th} instance of w and $in_{r_i}(j)$ be the value read by 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. $O_{wi} < T_w$.

Unlike the rate transition buffering scheme used by the Real-Time Workshop tools, we define communication to be 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/NLPR readers have higher/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 a λ-unit delay, M_1 readers communicate with a $(\lambda+1)$-unit delay, etc. Let κ represent the longest link delay associated with communication links, i.e. $\kappa = \max(p, \lambda + q)$.

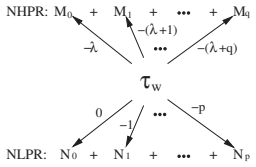

Figure 2. Comm. of Single Writer/Multiple Readers

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 based on the input values. Note that, in the middle of the figure, it is $i_{r_i}(j) = o_w(k)$ during simulation (under the SR zero-time semantics). The bottom of Figure 3 shows the possible

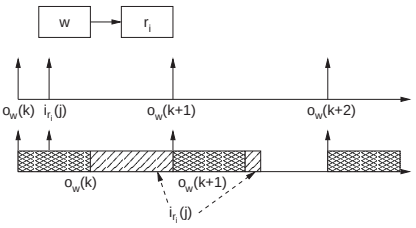

Figure 3. Data Integrity and Determinism Problems

problems with data transfer 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 thus 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 buffer-based 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. Integral 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.

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:

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

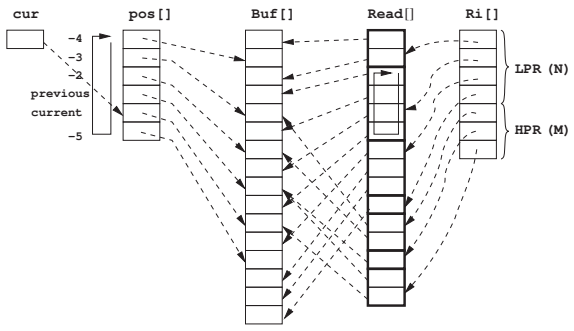

Figure 4. Based on Spatially-out-of-Order Writes

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.

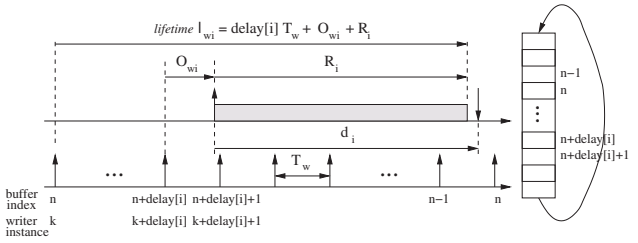

Figure 5. Based on Spatially-In-Order Writes

Figure 5 illustrates the basic idea about this mechanism. We define the 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:

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

Under this mechanism, the writer just keeps writing data into the next slot in a circular buffer at its own sampling rate. The good thing is that it does not require much bookkeeping to achieve constant execution time for finding a safe buffer slot for the writer. This mechanism is also known as 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. Furthermore, 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.

For convenience, notations are first summarized in Table 1. delay and IsHPR are defined for all SysNIP input ports of the tasks. Similarly, cur, prev, NLPR, WrtInit, and NB are defined for all SysNOP output ports in the system.

The total number of buffers required is simply the sum of buffer sizes for all writers:

$$\texttt{SysNB} = \sum_{1 \le o \le \texttt{SysNOP}} \texttt{NB}_{w_o}, \quad (4)$$

where $\texttt{NB}_{w_o}$ 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.

NT	number of tasks	pri	task priority
Buf[]	shared comm. buffer	SysNB	total buffer size
SysNIP	number of input ports	SysNOP	number of output ports
Read[i]	buffer slot currently used by reader i		
delay	link delay	IsHPR	relative priority
cur/prev	buffer slot with latest/immediate previous data		
NLPR	number of lower-priority readers		
WrtInit	initial output value	NB	# of buffers of writer

Table 1. Notations Used to Describe a System

4.1 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.

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. There are different ways to implement the FindFree() search algorithm used to find a safe buffer slot for the writer at its activation time. In the following paragraphs, we present the implementations for the DBP with a linear and a constant time search algorithm.

4.1.1 The Linear Time DBP

Figure 7 shows the data structures used for the DBP with a Linear Time FindFree(). 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,

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;`	`/* termination time */`
	`Read[i] = FREE;`
Higher priority Reader	
`/* activation time */`	`/* execution time */`
`HPR[i] = prev;`	`···`
	`··· = Buf[HPR[i]];`
	`···`

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

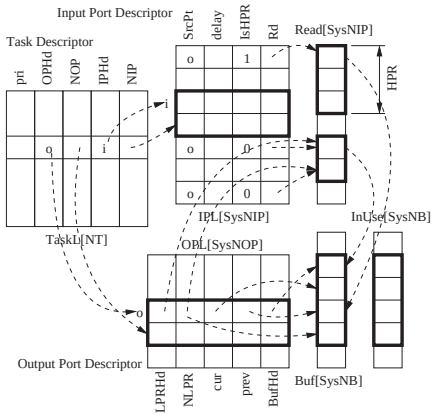

Figure 7. Data Structure for LTDBP

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 7. 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. This is slightly different from what is in [12], where the DBP is originally presented. All readers and writers share array `Buf[SysNB]`. Similarly, `BufHd` and `NB` specify a contiguous segment in `Buf[]` for a writer.

Figures 8, 9, and 10 give the complete LTDBP.

Data Structure		
`struct TaskEntry {`	`struct OPEntry {`	`struct IPEntry {`
`  char pri;`	`  char LPRHd;`	`  char SrcPt;`
`  char OPHd;`	`  char NLPR;`	`  char delay;`
`  char NOP;`	`  char cur;`	`  char IsHPR;`
`  char IPHd;`	`  char prev;`	`  char Rd;`
`  char NIP;`	`  char BufHd;`	`} IPL[SysNIP];`
`} TaskL[NT];`	`} OPL[SysNOP];`	
`char Read[SysNIP];`	`message WrtInit[SysNOP];`	
`char InUse[SysNB];`	`message Buf[SysNB];`	

Initialization

```
/* OPL[].BufHd/LPRHd/cur/prev, buffer */
OPL[0].BufHd = OPL[0].cur = OPL[0].prev = 0;
topLPR = 0;
compLPRHd(0, &topLPR);
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];
  compLPRHd(i, &topLPR);
}
/* IPL[].Rd and Read[]; assume given IPL[].IsHPR */
for (i = 0; i < SysNIP; i++)
  temp1[i] = 0;
j = SysNIP - 1;
for (i = 0; i < SysNIP; i++) {
  if (IPL[i].IsHPR == 1) {
    IPL[i].Rd = j;
    j = j - 1;
  } else { /* contiguous Read[] for LPR */
    IPL[i].Rd = tmp1[idx1] + OPL[idx1].LPRHd;
    tmp1[idx1]++;
  }
  Read[i] = -1;
}
```

```
compLPRHd(char i, char *topLPR) {
  if (OPL[i].NLPR == 0) {
    OPL[i].LPRHd = -1;
  } else {
    OPL[i].LPRHd = *topLPR;
    *topLPR = OPL[i].LPRHd + OPL[i].NLPR;
} }
```

Figure 8. DS Declaration and Initialization for LTDBP

The data structure declaration and the initialization code is shown in Figure 8. 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.

Figure 9 shows the code for application tasks when using the LTDBP . 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 `FindFree()` function is used, the execution time of each writer at the kernel level depends in the worst case on the buffer size. The indexing code for each reader that needs to be executed at the kernel level clearly finishes in constant time for each

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

/* each reader i */
j = IPL[i].SrcPt;
k = IPL[i].Rd;
if (IPL[i].delay)
  Read[k] = OPL[j].prev;
else
  Read[k] = OPL[j].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 9. Application Tasks for LTDBP

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. 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 `prev` and `cur`.

```
char FndFrLTDBP(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 != -1)
      InUse[j] = 1;
  }
  for(k=BufHd;InUse[k];k++)
    ;
  return k;
} /* O(NB) */
```

Figure 10. `FindFree()` for LTDBP

Figure 10 shows the `FndFrLTDBP()`. It takes as argument the writer's index, which guides the search algorithm to use the data belonging to this writer.

Memory requirement is shown in Table 2.

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

Table 2. LTDBP Memory Requirement

4.1.2 The Constant Time DBP

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.

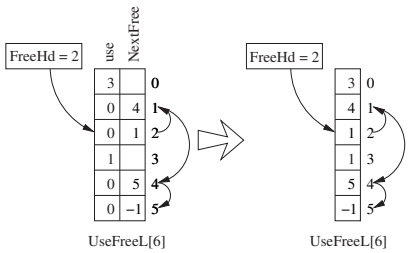

Figure 11. A Use Free List Data Structure

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 11. 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 11, 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. We 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, `FndFrCTDBP()`.

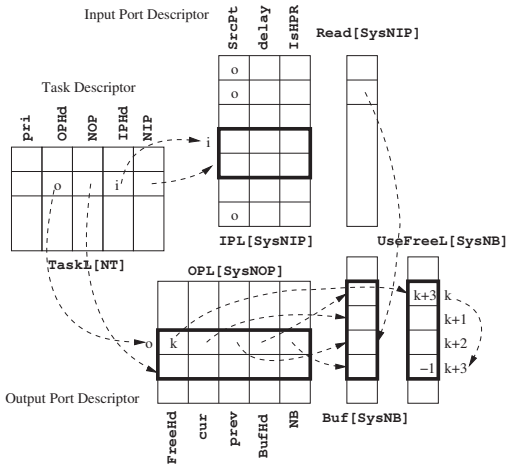

Figure 12. Data Structure for CTDBP

Figure 12 shows the data structures used for the DBP with a Constant Time `FindFree()`. Notice that the task descriptor is exactly the same as its counterpart in Figure 7. Since the `FndFrCTDBP()` 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. Note that lower-priority readers are not necessarily mapped onto a contiguous section in `Read[]`. 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 12.

```
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, cur, prev, & buffer init as Fig 8 */
/* initialization of use free list */
for (i = 0; i < SysNOP; i++) {
  UseFreeL[OPL[i].BufHd] = 2; /* not free */
  OPL[i].FreeHd = OPL[i].BufHd + 1;
  for (j = 1; 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 13. DS Declaration and Initialization for CTDBP

Figure 13 shows the data structure declaration and the corresponding initialization. Similar to the LTDBP in Figure 8, 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. For simplicity, a contiguous buffer segment is associated with a writer.

```
/* activation time */
/* each writer i */
UseDec(i, OPL[i].prev);
OPL[i].prev = OPL[i].cur;
OPL[i].cur = FndFrCTDBP(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;
  UseDec(t2,t1);
}
```

Figure 14. Application Tasks for CTDBP

Similar to the LTDBP, this communication protocol is executed at the kernel level and the application level. Comparing the code for the writer and readers with their counterparts in Figure 9, 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.

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.

```
char FndFrCTDBP(char i) {
 t = OPL[i].FreeHd;
 OPL[i].FreeHd=UseFreeL[t];
 return t;
} /* O(1) */

void UseDec(char i,char j){
  UseFreeL[j]--;
  if(UseFreeL[j] == 0) {
    freeHd = OPL[i].FreeHd;
    if(freeHd == -1) {
      UseFreeL[j] = -1;
      OPL[i].FreeHd = j;
    } else {
      UseFreeL[j] = freeHd;
      OPL[i].FreeHd = j;
} } }
```

Figure 15. `FindFree()` for CTDBP

Figure 14 shows the constant time search algorithm. It takes in the writer's index as input to return the head of the writer's free list. During execution, the `FndFrCTDBP()` 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.

The memory consumption of the implementation of the CTDBP is shown in Table 3.

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

Table 3. CTDBP Memory Requirement

4.2 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). Figure 16 shows the used data structures. The size of the shared buffer is computed using Equation 4. Similar to the DBPs in Section 4.1, 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 16 are much simpler than those used for the DBPs in Figures 7 and 12.

Figures 17 and 18 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 17. Each writer is assigned a continuous segment of `Buf[]`, which is identified by the pair

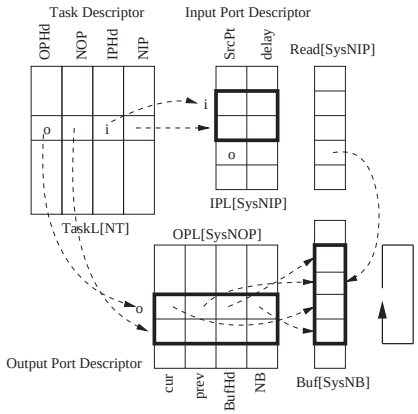

Figure 16. Data Structure for TCCP

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 17. DS Declaration and Initialization for TCCP

of `BufHd` and `NB` as shown in Figure 16. `cur`, `prev`, the initial output value are initialized for each writer. As shown in Figure 18, the buffer indexing procedure performs buffer index assignments at kernel level for each individual reader or writer. 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 a task may have multiple ports, it needs to process all of them.

Since the writer writes data into a circular buffer in a spatially-in-order manner, the `FndFrTCCP()` simply takes as argument the index of the writer, increments the `cur`, take the modulus operation with respect to the size of the buffer and then return the remainder as the new `cur` for the writer.

The memory consumption of the extended implementation of the TCCP is shown in Table 4.

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

Table 4. TCCP Memory Requirement

`/* activation time */`
`/* each writer i */`
`OPL[i].prev = OPL[i].cur;`
`OPL[i].cur = FndFrTCCP(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 FndFrTCCP(char idx) {`
	`  return (OPL[idx].cur+1)\`
	`  % OPL[idx].NB;`
	`} /* O(1) */`

Figure 18. Application Tasks and `FindFree()` for TCCP

4.3 Comparison of (LT/CT)DBP and TCCP

Compared with the `FindFree()` for LTDBP, though the `FindFree()` for CTDBP executes 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 based on the number of lower-priority reader tasks.

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 [13], and microITRON [14] have been developed. In this paper, we choose to focus on OSEK. The OSEK 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 5. 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 5. 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 [15] 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 application tasks and it cannot be preempted by ISRs in category 2. Table 6 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	Activate/terminate/chain/state reference
Scheduling	Non/full/mixed preemptive; RR same level
Multiple Activation	BCC2 tasks and basic tasks in ECC2
Memory Management	No virtual memory(MMU)/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 6. 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 19 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.

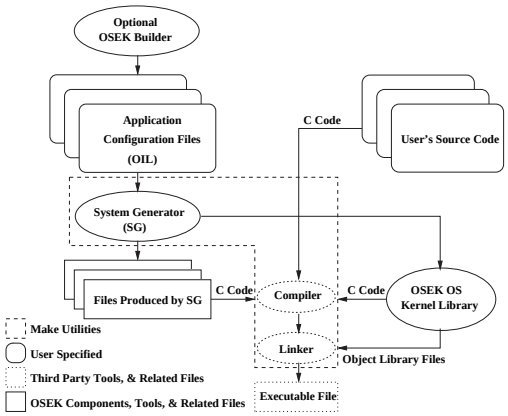

Figure 19. Application Development Process

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

Object	Mandatory	Standard Attribute	Std Reference
CPU	yes	-	-
OS	yes (= 1)	STATUS; USERESSCHEDULE; USEGETSERVICEID; Hooks; USEPARAMETERACCESS	-
APPMODE	yes ($\geq$ 1)	-	-
TASK	yes ($\geq$ 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 7. OIL Objects and Their Properties

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 5, 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 dispatcher defines the proper buffer indices for writers and readers upon

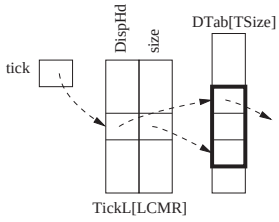

Figure 20. Task Dispatcher Data Structure

```
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 21. Task Dispatcher Declaration and Initialization

their activation. This implies that the dispatcher executes at the kernel level. To handle all different periods of application tasks, the dispatcher executes at a 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 dispatcher are shown in Figure 20 and the corresponding declaration is shown in Figure 21. 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 21. The entries of DTab[] are used to index the tasks in the task descriptor presented earlier.

The right column in Figure 21 is the initialization of the data structures used for the 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 dispatcher will be discussed later in this section. Recall that the data structures of the communication protocols discussed in Section 4 also need to be initialized to obtain correct execution semantics. The initialization code is shown in Figure 22. This can be implemented as an initialization OSEK task; alternatively these can be done in the main function of the application during the system startup. Considering the initialization is only performed once during the system startup and an OSEK task implementation may affect the scheduler's performance, we choose to perform initialization in the main function of the application.

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 relative priority, IsHPR, in the task descriptor needs to be computed at system startup and stored for later use. Besides initializing data structures, a cyclic alarm called dispAlarm, associated with the dispatcher is set up during system startup.

```
... /* init DS required by protocol */
... /* init dispatcher as in Fig 21 */
/* set up the alarm for dispatcher */
SetRelAlarm(dispAlarm, 0, GCDR);
```

Figure 22. Initialization at System Startup

In summary, for a portable OSEK implementation under BCC1 consists of a dispatcher and one or more application tasks. All OSEK tasks need to be declared as shown in Figure 23 before being used.

```
DeclareTask(AppTask_i);
...
```

```
DeclareTask(dispatcher);
```

Figure 23. OSEK Task Declaration

6.2 Task Implementation in C Code

In Section 4, three SR semantics preserving protocols are presented: LTDBP, CTDBP, and TCCP . In this section, we first present common data structures and the definition of the task dispatcher for an OSEK BCC1 implementation of applications using these protocols. The declaration for common data structures used in implementations is shown in Figure 24.

```
#DEFINE NT X
#DEFINE LCMR X
#DEFINE GCDR X
```

```
#DEFINE SysNIP X
#DEFINE SysNOP X
```

```
#DEFINE TSize X
#DEFINE SysNB X
```

```
message Buf[SysNB];
message WrtInit[SysNOP] = {X, ...};
```

```
char Read[SysNIP];
```

Figure 24. Common DS Declaration

Figure 25 shows the general structure of the task dispatcher. 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. According to a dispatch table precomputed during system startup, 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 writing/reading indices for each input/output port that the task has and then the dispatcher activates the task using the API ActivateTask. At the end of the definition, the dispatcher calls TerminateTask for self-termination.

```
1 TASK(dispatcher) {
2   tick = (tick+1) % LCMR;
3   if (TickL[tick].DispHd != -1) {
4     for (k=0; k<TickL[tick].size; k++) {
5       idx = DTab[k+TickL[tick].DispHd]; /* taskID */
6       for (i=0; i<TaskL[idx].NOP; i++) {/* writer */
7         idx2 = TaskL[idx].OPHd + i;
8         ··· /* kernel level writer code */
9     } }
10    for (k=0; k<TickL[tick].size; k++) {
11      idx = DTab[k+TickL[tick].DispHd];
12      for (i=0; i<TaskL[idx].NIP; i++) {/* reader */
13        idx2 = TaskL[idx].IPHd + i;
14        ··· /* kernel level reader code */
15      }
16      ActivateTask(idx);
17  } }
18  TerminateTask();
19 }
```

Figure 25. Functionality of Task Dispatcher

6.2.1 Implementation of the DBP

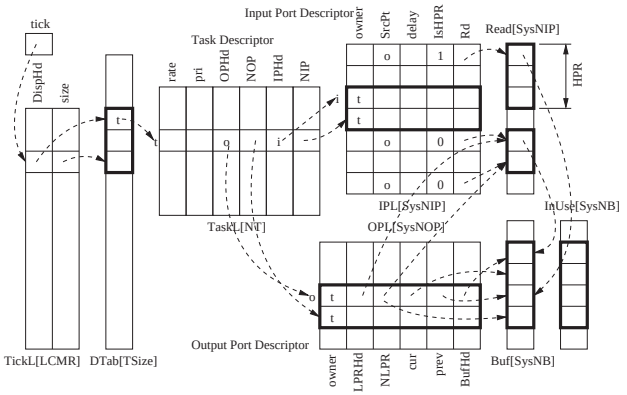

Figure 26. LTDBP Implementation DS

Implementation of LTDBP Figure 26 shows the data structures needed for our OSEK implementation of applications using the LTDBP. It combines the data structures in Figures 7 and 20 with slight modification. Comparing with the task descriptor in Figure 7, fields `rate`, `pri`, and `owner` are added into both the input port and output port descriptors.

Figures 27, 24, and 21 together show the complete corresponding data structure declaration. The implementation of the dispatcher has been shown in Figure 25. The dispatcher performs kernel-level operations in Figure 9 based on whether it is writer or reader. Specifically it calls `FndFrLTDBP()` 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 28 shows the OSEK implementation of the communication protocol implemented at the application level, 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 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`. Note that our implementation guarantees that the operations in the termination code are executed with a correct execution semantics since single memory operation is atomic.

```
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 27. DS Declaration for LTDBP

```
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 28. OSEK Implementation of App. Task for LTDBP

The initialization code required in Figure 22 for the LTDBP is from Figure 8 and the auxiliary field `IsHPR` is initialized as shown in Figure 29.

```
for (j=0; j<SysNIP; j++) { /* init IPL[].IsHPR */
  owner = OPL[IPL[j].SrcPt].owner;
  if (TaskL[IPL[j].owner].pri > TaskL[owner].pri)
    IPL[j].IsHPR = 1;
  else
    IPL[j].IsHPR = 0;
}
```

Figure 29. Part of Initialization for the LTDBP

Implementation of CTDBP In this section, we present an OSEK application implementation when using the CTDBP shown in Figure 13. Figure 30 illustrates the data structures used in the implementation. It combines the data structures for the task dispatcher in Figure 20 and the corresponding data structures for the CTDBP in Figure 12. Similar to the implementation for applications with LTDBP, besides `rate` and `owner`, a new field called `done` is is further introduced

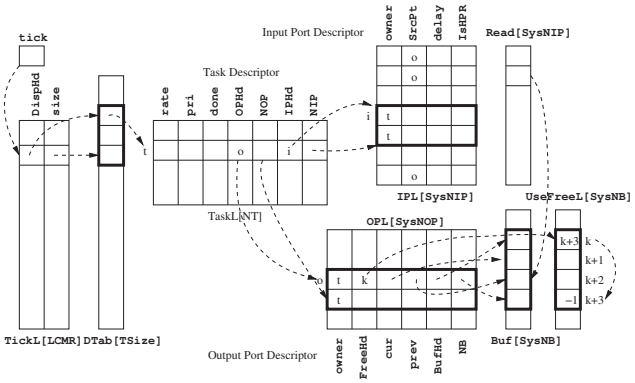

Figure 30. CTDBP Implementation DS

for the purpose of guaranteeing when the critical section at the end of lower-priority readers needs to be executed. The corresponding data structure declaration is shown in Figures 31, 24, and 21.

```
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},
  ···};
char UseFreeL[SysNB];
```

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

Figure 31. DS Declaration for CTDBP

The task dispatcher shares the same structure as shown in Figure 25: 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 is from Figure 14.

```
TASK (AppTask_i) {
  TaskL[i].done = 0;
  ···
  /* each writer k */
  Buf[OPL[k].cur] = ···
  ···
  /* each reader k */
  ··· = Buf[Read[k]];
  ···
  TaskL[i].done = 1;
  /* atomic hook code */
  TerminateTask();
}
```

```
void PostTaskHook(void) {
  char i,id,j,k,nip,t1,t2;
  GetTaskID(&id);
  if (id>0 && id<=NT) {
    i = id - 1;
    if (TaskL[i].done) {
      nip = TaskL[i].NIP;
      for (j=0; j<nip; j++){
        k = j+TaskL[i].IPHd;
        ···/* CS of Fig 14 */
      }
    }
} }
```

Figure 32. OSEK Implementation of App. Task for CTDBP

Figure 32 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 4.1.2, we know that the constant execution time of the search algorithm `FndFrCTDBP()` is obtained through using a use free list. Lower-priority readers 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, OSEK APIs `SuspendOSInterrupts` and `ResumeOSInterrupts` could be used to gain exclusive access to the shared use free list.

The initialization required for this OSEK application implementation performs the same functionalities as its counterpart in Figures 22 and 29. The only difference is that the initialization code for the CTDBP is from Figure 13.

6.2.2 Implementation of TCCP

Figure 33 shows the data structures used for the OSEK implementation. It is the combination of the data structures in Figures 20 and 16 with field `rate` added to the task descriptor. The complete corresponding declaration is shown in Figure 34, 24, and 21.

The task dispatcher performs the same functionality as shown in Figure 25 and the corresponding kernel-level application code is from Figure 18.

The definition of application tasks shown in Figure 35 shares the same structure with its counterpart

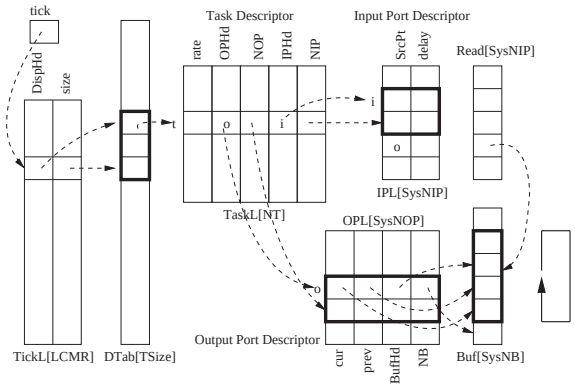

Figure 33. TCCP Implementation DS

```
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 34. DS Declaration for TCCP

```
TASK (AppTask_i) {
  ···
  /* each writer k */
  Buf[OPL[k].cur] = ···
  ···
```

```
  /* each reader k */
  ··· = Buf[Read[k]];
  ···
  TerminateTask();
}
```

Figure 35. OSEK Implementation of App. Task for TCCP

shown in Figure 28 and it is even simpler because there is no bookkeeping termination code in the protocol.

The initialization required for this OSEK application implementation has a similar but simpler structure compared with those of the DBPs since the field `IsHPR` is not needed. The initialization code for the TCCP is from Figure 17.

6.2.3 OSEK Implementation Comparison

In this section, we compare the implementations presented earlier. Table 8 shows the memory requirements for the application implementations with different versions of SR semantics preserving protocols. For applications with single-port tasks, the CTDBP and the LTDBP requires about the same amount of chars 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.

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

protocol	char	message
LTDBP	6×(NT+SysNOP+SysNIP)+ 2×LCMR+SysNB+TSize+1	SysNB+SysNOP
CTDBP	7×NT+6×SysNOP+5×SysNIP+ 2×LCMR+SysNB+TSize+1	SysNB+SysNOP
TCCP	5×NT+4×SysNOP+3×SysNIP+ 2×LCMR+TSize+1	$SysNB_T$+SysNOP

Table 8. Memory Requirement Comparison

`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, 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;
  };
  /* 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 36. OIL Configuration File

Figure 36 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 dispatcher, therefore the attribute `AUTOSTART` is set as `FALSE`. 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 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 the dispatcher through setting its attribute `ACTION` as `ACTIVATETASK`. Finally the alarm's `AUTOSTART` attribute is set as `TRUE` and explicitly associating the dispatcher that needs to be activated. The period of `dispAlarm` is assigned as `GCDR`.

When the constant time `FndFrCTDBP()` 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 36.

7 Performance Evaluation under the PICos18

In Section 6, portable OSEK implementations of the SR semantics preserving communication protocols are presented and their temporal properties are analyzed qualitatively. In this section, we evaluate the performance properties of the implementations under a specific OSEK-compliant RTOS: PICos18.

7.1 What is PICos18

The PICos18 [16], developed by Pragmatec SARL Company [17] in France, is a multi-task, preemptive, and real-time kernel that fully conforms to the OSEK standard for the PIC18 family of the Microchip Technology Inc [18]. Being real-time, the kernel guarantees a deterministic latency time for task switching from the current one to another that is more urgent. It is a piece of free software and distributed under the terms of the GNU General Public License [19].

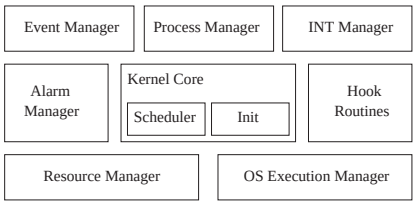

Figure 37. The PICos18 Kernel

Figure 37 shows the components implemented in the PICos18 Kernel. The kernel core is composed of scheduler and init. The init is to initialize the kernel after the `StartOS` kernel service called from the main function of the application. The kernel scheduler supports concurrent task execution through context switches: for context saving, the context of the task in the special file registers, the hardware stack, the dedicated RAM area for math and temporary data are saved into task's RAM area; for context restoring, the task's context in its RAM area is copied back.

The kernel services for task management are implemented in the process manager, which is in charge of changing/querying the state of a task and querying the identifier of the currently running task. The interrupt manager handles the set of kernel services for interrupt disabling/enabling as well as suspending/resuming by resetting/setting corresponding control bits in the interrupt control register. The resource manager implements the OSEK Priority Ceiling Protocol (PCP) for shared resource access by dynamically assigning and restoring task's priority to avoid priority inversion and deadlock. The event manager implements the set of kernel services for event posting/clearing/waiting/reading through approximate event masks. It manages activations of extended tasks when some event(s) waited by an extended task set by another task or an ISR of category 2. The alarm manager is used to turn on/off an alarm as needed and to configure an active alarm accordingly from the application layer. The alarms are periodically updated based on the system timer (TIMER0) interrupt inside the kernel. The OS execution control is implemented in the OS execution manager, which manages active application mode querying as well as kernel starting and stopping. The hook routines function like execution of the functionalities from designers at the kernel level. In the currently available version of the PICos18 (Version 10), it is only implemented to support `StartupHook`, `PreTaskHook`, and `PostTaskHook`.

These kernel services can be utilized by application through calling corresponding standard APIs. In the PICos18, the kernel core is coded in assembly language and the other components are programmed in C language. The characteristics of the PICos18 Version 2.10 are summarized in Table 9.

Kernel category	Multi-task and preemptive
Target processors	Family of the PIC18
Kernel size (ROM/RAM)	< 1 kBytes/7 Bytes
Service size (ROM/RAM)	4 KBytes/121 Bytes
Hardware stack	31 function calls
Software stack size	64, 128, 256 Bytes
Latency of the kernel	140μsec (CPU frequency:40MHz)
Maximum number of tasks	16
Maximum event number per task	8
Number of priorities	16 (0-15)
Number of software timers	Limit to the size of RAM

Table 9. Features of the PICos18 Version 2.10

7.2 Execution Microprocessor: PIC18F452

During Execution, only one task can have access to the processing unit, that is the exclusive access to the processor, to the RAM memory, and to the hardware stack. The PICos18 is designed and implemented for the PIC18 family of microprocessors from the Microchip Technology Inc. Particularly, the PIC18F452 [20] processor chosen as the underlying execution platform in this study. The PIC18F452 processor is type of RISC and it is an enhanced flash microcontroller with a high performance up to 10MIPS (Millions of Instructions Per Second) operations. The features of the PIC18F452 are shown in Table 10. The PIC18F452 processor has a C compiler optimized Instruction Set Architecture (ISA) with 16-bit wide instructions and it supports priority levels for interrupts. Its data path has a width of 8 bits. Three memory blocks include program memory, data RAM, and data EEPROM. It has 32 KBytes and 1.5 KBytes linearly addressable program memory and data memory as shown in Figure 38. The data memory is implemented as static RAM and it contains Special Function Registers (SFRs) and General Purpose Registers (GPRs). Separate buses are used for data and program memory and therefore concurrent data and instruction accesses are supported. The RESET vector stars at 0000h and the interrupt vectors start at 0008h and 0018h for high-priority and low-priority interrupts, respectively.

The PIC18 processors manage the hardware stack that is dedicated for function calls via the PUSH/POP instructions. The return address stack is a piece of RAM with 31 words with a width of 21 bits. It supports any combination of up to 31 program calls and interrupts. Upon execution of CALL/RCALL or acknowledgement of an interrupt, the stack pointer is first incremented and the value of the Program Counter (PC) is pushed onto the stack slot pointed by the stack pointer, while upon execution of RETURN/RETLW/RETFIE, the PC value is popped off the stack and the stack pointer is decremented accordingly.

Device	On-Chip Program Mem FLASH (Bytes)	On-Chip Program Mem # Single Word Instr	On-Chip RAM (Bytes)	Data EEPROM (Bytes)	Interrupt Source	Instr Set
PIC18F452	32K	16K	1.5K	256	18	75

Table 10. Features of the PIC18F452 Microcontroller

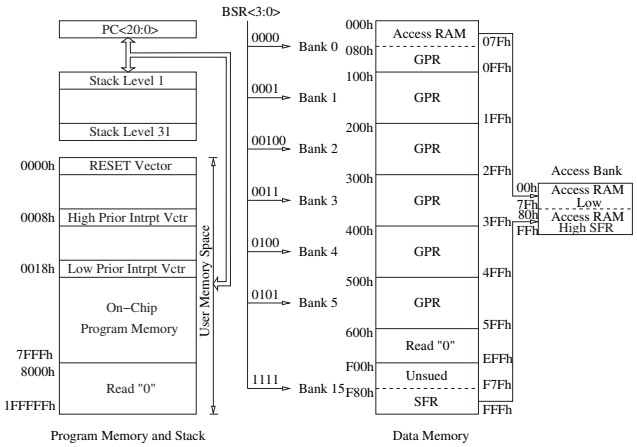

Figure 38. Memory of PIC18F452

Based on the clock input, four non-overlapping quadrature clocks (Q1, Q2, Q3, and Q4) are generated via a clock divider as shown in Figure 39. An Instruction Cycle (IC) is defined as the four Q cycles (Q1, Q2, Q3, and Q4). The program counter is incremented during Q1 and the instruction is fetched from the program memory and latched into the instruction register during Q4. The instruction decoding and execution take place during the next Q1 to Q4. Effectively, each instruction takes one IC due to the pipelining.

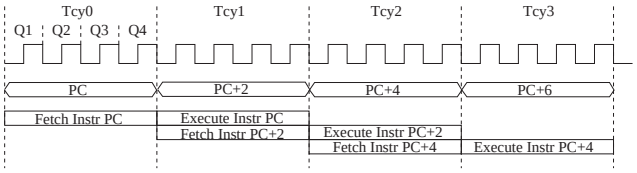

Figure 39. Instruction Pipeline of PIC18F452

7.3 Application Development Environment

The used application development environment is the Windows-based Microchip compiler toolsuite: MPLAB Integrated Development Environment (IDE). The MAPLAB IDE consists of the MPASM Assembler [21], the MCC18 compiler [22], and the MPLINK linker [21]. The MPASM and the MCC18 transform assembly files and C files into relocatable object files,

respectively. The MPLIB Object Librarian manages pre-compiled code to be used with the MPLINK object linker. The MPLINK object linker combines all the object files to generate a unique HEX file that can be loaded into a PIC18 processor.

Then the compiled application can be analyzed through simulation/emulation. The MPLAB SIM simulator support code development in a PC-hosted environment and microcontroller simulation at the instruction level. Furthermore, the MPLAB In-Circuit Emulator (ICE) 2000 supports enhanced features such as trace, trigger, and data monitoring. Finally, the MPLAB In-Circuit Debugger (ICD) is a powerful runtime development tool with the in-circuit debugging capability.

7.4 Performance Measurements

In this section, we study the temporal properties of the different implementation versions of the SR semantics preserving communication protocols. The PIC18F452 [20] processor is used as the underlying execution platform. With the PIC18F452 processor as the execution platform, when the processor frequency is 40MHz, the CPU performance is 10MIPS (Millions of Instructions Per Second) and one instruction cycle takes $0.1\mu sec$. The software system timer used to manage alarms and counters in the PICos18 has a cycle of $10msec$.

Analysis [23] shows that the cost to save context is `(308+12×ItemStk)ICs`, where `ItemStk` is the number of return addresses on the hardware stack. Clearly this is application dependent. Similarly, the cost to restore context is `(316+11×ItemStk+22×IterCur)ICs` instruction cycles, where `IterCur` is the number of iterations that needed to locate the task from the TCTs. Clearly it depends on the priority of the newly chosen running task. If kernel service `Schedule` is called from an ISR, its execution time is 11ICs. Otherwise, for the USER mode, its execution time is `(681+12×ItemCur+36×TskSmPrioCur+6×IterCur+PostTskHk+32×IterNxt+11×ItemNxt+PreTskHk)ICs`, where `ItemCur` and `ItemNxt` are the number of return addresses on the hardware stack of the currently running task and the new running task that needs to be restored, respectively, `TskSmPrioCur` is the number of tasks that have the same priority as the currently running task, `IterNxt` is the number of iterations that needed to locate the new running task from the TCTs, `PostTskHk` and `PreTskHk` are cost due to the `PostTaskHook` and the `PreTaskHook`, respectively; for the non-USER mode, its execution is shorter by the time needed for context saving because no task context needs to be saved. The contributions to its worst case performance are from saving context of the old running task, choosing the next running task, restoring context of the newly chosen running task, and possible `PreTaskHook`/`PostTaskHook`. When all application tasks have unique priorities, there is no hook involved, and there are two return addresses on the hardware stack to be saved/restored, the latency of the kernel scheduler is `1395ICs`, which is about $140\mu sec$ and occupies about 1.4% of a system tick.

From Section 6.2, we know that the implementation of the CTDBP depends on the `PostTaskHook` and this may increase the size of the footprint. We compiled the kernel to generate the library twice and the library sizes are 123661 and 123754 Bytes for without and with the `PostTaskHook` option, respectively. The 93-byte difference gives the absolute size overhead due to turning on the `PostTaskHook`. Comparison with Table 9 shows that this overhead is small.

Section 6.2 illustrates that the implementations of application tasks are slightly different to achieve the same functionality. For the LTDBP, an application task needs to flag its termination for all its lower priority readers when it finishes. For the CTDBP, at the very beginning, an application initializes its flag `done` as false, while it turns the flag to true upon termination. The update on the shared use free list for all its lower priority readers is fulfilled through calling a post task hook from the kernel. For the TCCP, there is no extra operation that needs to be executed to guarantee correct functionality. Another observation is that the dispatchers share exactly the same structure. The only differences among them are the way to assign legitimate writing and reading indices for application tasks.

To ease comparison, we dissect the dispatcher as follows. The first segment accounts for the minimum cost being a kernel level dispatcher, whose cost is represented by $C_{k,d}$. Specifically it consists of three parts: the update of `tick` in Line 2 of Figure 25 ($C_{k,d,1}$), the check of the unsatisfaction of the `if` condition in Line 3 ($C_{k,d,2}$), and the call to `TerminateTask` in Line 18 ($C_{k,d,3}$). Obviously, the execution time of this segment corresponds to the temporal cost when there is no task that needs to be activated at some `tick`.

The second segment of the dispatcher corresponds to the minimum cost associated with a task that needs to be activated at some `tick`, which is denoted by $C_{k,\tau}$. Coincidentally, it is also composed of three portions: the setup and false iteration condition checking of the first/second `for` loop for dealing with writers/readers in Lines (4-7)/(10-13) ($C_{k,\tau,1}$)/($C_{k,\tau,2}$), and the call to `ActivateTask` in Line 16 ($C_{k,\tau,3}$). Clearly the temporal cost of the second segment is due to a task with a single writer and reader that needs to activated. Note that it excludes the cost for index assignments.

The third segment of the dispatcher accounts for index assignment for a single writer/reader in Line 8/14,

which is denoted by $C_{k,w}/C_{k,r}$. In the following subsections, we measure and compare the temporal characteristics of the segments identified above.

7.4.1 Measurements of the LTDBP

When using the LTDBP, the measurements of the breakdown of the dispatcher are shown in Tables 11 and 12. Note API `ActivateTask` takes `(809+25×dis)`

		$C_{k,i,1}$	$C_{k,i,2}$	$C_{k,i,3}$
$i = d$		113	17	115
$i = \tau$	LTDBP/TCCP	127	127	809+25×dis
	CTDBP	127	127	875+25×dis

Table 11. Characterization Parameter Measurement (ICs)

	$C_{k,d}$	$C_{k,\tau}$	$C_{k,w}$	$C_{k,r}$
LTDBP	245	1063+25×dis	$(398+110\times NLPR)^{(wc)}$	$177^{(wc)}$
CTDBP	245	1129+25×dis	$(365)^{(wc)}$	$191^{(wc)}$
TCCP	245	1063+25×dis	265	$133^{(wc)}$

Table 12. Dispatcher Performance Evaluation (ICs)

ICs, where `dis` is the distance of the activated task to head of the task queue that has been ordered in a non-increasing priority order. Note that the dispatcher has the highest priority and its `dis` value is 1. There are two return addresses that need to be saved and restored on the hardware stack during context switch of the dispatcher because the maximum overlapping call for the dispatcher is 2. When measuring the running time of `FindFree()` for LTDBP, we assumed that the number of buffers needed for a writer is be `NB=NLPR+2`.

Measurements show that the cost to flag its termination for lower priority readers of an application task is `(36+123×NLPR+91×NHPR)ICs`. Recall that `NLPR` and `NHPR` are the number of lower and higher priority readers a writer may have, respectively.

7.4.2 Measurements of the CTDBP

When using the CTDBP, the execution time for each segment of the dispatcher when the `PostTaskHook` is shown in Tables 11 and 12. Note API `ActivateTask` takes `(875+25×dis)ICs`, which includes the overhead from a call to the hook routine in the scheduler.

For the CTDBP, measurements show that the costs to reset/set the flag `done` at the very beginning/end are `23ICs` and `25ICs`, respectively.

The `PostTaskHook` option is turned on to achieve atomicity for the critical section of the termination code for a task to update the use free list(s) for all its lower priority readers it may have. Analysis on the `PostTaskHook` routine shows its cost is `66ICs` if no application task executed right before calling the hook routine; its cost is `100ICs` if some application task executed and had not terminated yet when the call was made; its cost is `(138+233×NLPR+96×NHPR)ICs` in the worst case if some application task was just terminated and it triggers the call to the hook routine to perform use free list update at the kernel level for the lower priority readers that a task may have. Note that the dispatcher task is not an application task and the overhead on running time introduced to it by a call to the hook routine is always `66ICs`.

The execution of the hook routine after an application task's termination is needed. However, it is called during each context switch during system execution. We analyze the temporal overhead due to calling the routine when the functionality in the hook is not needed. Measurements show that the overhead of calling the hook routine is `100ICs` in the worst case, which is $10\mu sec$. This means that the latency of the kernel scheduler increases to $150\mu sec$ from $140\mu sec$. The relative overhead increase is 7.1% and now the task context switch occupies about 1.5% of a system tick.

7.4.3 Measurements of the TCCP

Note that there is no bookkeeping code in the implementation of application tasks when using the TCCP. The measurements of temporal characteristics of the breakdown of the dispatcher are shown in Tables 11 and 12.

7.5 Performance Comparison

Table 13 summarizes the performance comparison of application tasks when using the LTDBP, CTDBP, and TCCP. Clearly an application task with the TCCP is the fastest, while an application task with the LTDBP is the slowest. An application task with the CTDBP executes also fast, but the kernel performs the required functionality in a post task hook routine on behalf of the application task. The measurements show that the amount of time taken by the `PostTaskHook` routine under the CTDBP is longer than the corresponding version under the LTDBP by `(150+110×NLPR+5×NHPR)ICs`.

		App Task	PostTaskHook
DBP	LT	36+123×NLPR+91×NHPR	0
	CT	48	138+233×NLPR+96×NHPR
TCCP		0	0

Table 13. Application Task Comparison (ICs)

Table 12 shows the performance comparison of dispatchers when using the LTDBP, CTDBP, and TCCP. Because all three versions of the dispatcher shares exactly the same structure, their $C_{k,d}$ values are all `245ICs`. Because the LTDBP and the TCCP do not need the `PostTaskHook` routine, their $C_{k,\tau}$ values are both `(1063+25×dis)ICs`. The $C_{k,\tau}$ value for the CTDBP is larger by `66ICs`, which is due to the call to the `PostTaskHook` routine in the kernel scheduler when

calling `ActivateTask` inside the dispatcher. This overhead is about 6%.

The $C_{k,w}$ and $C_{k,r}$ characterize the temporal cost of the communication protocol used. To assign a legitimate index for a writer, the TCCP takes the least amount of time and the LTDBP takes the most amount of time. Consider a writer that has one lower priority reader, the LTDBP needs `508ICs`, which is about 92% and 39% more than the TCCP and the CTDBP, respectively. Similarly, to assign a legitimate index for a reader, the TCCP takes the least amount of time. Unlike to the writing index assignment, the CTDBP takes the most amount of time, which is 44% and 8% more than the TCCP and the LTDBP, respectively. In summary, the TCCP is the fastest among the three protocols we presented. Generally, it is difficult to compare the LTDBP and the CTDBP. The measurements in Table 12 show that the CTDBP may be better than the LTDBP if each writer has several lower priority readers. However the extra overhead on the context switch due to the `PostTaskHook` routine may also have a big impact on the system schedulability especially when the system tick is small compared with the processor clock period and the number of task preemptions is big. This is because the post task hook routine is called for each single context switch. In our study, the system tick is $10msec$, the processor clock period is $0.025\mu sec$, and the context switch cost with post task hook is $150\mu sec$. The ratio of the system timer to the processor clock is 400,000, which means that the context switch due to system timer interrupt is not big. As discussed in Section 7.4.2, the context switch overhead due to the system timer interrupt is 1.5% of a system tick.

Furthermore, when taking into account the memory requirements for different protocols, a proper communication protocol is highly dependant upon the temporal properties of the system. An optimization problem [24] has been studied in parallel with this work.

8 Conclusions

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()`. 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 the 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 footprint overhead due to the PostTaskHook in the final image of the CTDBP is 93 Bytes, which is small compared to the size of the kernel and services.

To study the temporal characteristics of the different versions of the implementations, we measured the temporal costs under the PICos18, which is a OSEK-compliant RTOS. Without the PostTaskHook, the overhead of a context switch is about $75\mu sec$. Turning on the PostTaskHook routine increases the context switch cost by 100 instruction cycles, which is $10\mu sec$ if the processor frequency is 40MHz. A dissection analysis on the dispatcher shows that the TCCP is the fastest for assigning indices for a writer and a reader. The CTDBP is faster on assigning indices for writers than the LTDBP and the LTDBP is faster on assigning indices for reader than the CTDBP. Taking into account the overhead due to the post task hook, the cons and pros of the LTDBP and CTDBP are application dependent. An optimization on memory consumption with consideration of timing constraint is desired and has been investigated for real-time applications.

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 prob-

lem," 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] G. Wang, M. D. Natale, and A. L. Sangiovanni-Vincentelli, "An osek/vdx implementation of synchronous reactive semantics preserving communication protocols," in *OSPERT 2007: Workshop on Operating Systems Platforms for Embedded Real-Time applications*, (Pisa, Italy), July 2007.

[12] 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.

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

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

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

[16] Pragmatec SARL Company, *PICos18, Version 2.10*. web page: http://www.picos18.com.

[17] Pragmatec SARL Inc. web page: `http://www.pragmatec.net`.

[18] Microchip Technology Inc. web page: `http://www.microchip.com`.

[19] GNU, "The general public license.." available at `www.gnu.org/copyleft/gpl.html`.

[20] Microchip Technology Inc., *PIC18FXX2 Data Sheet*. web page: http://www.microchip.com.

[21] Microchip Technology Inc., *MPASM Assembler, MPLINK Object Linker, MPLIB Object Librarian User's Guide*. web page: http://www.microchip.com.

[22] Microchip Technology Inc., *MPLAB C18 C Compiler User's Guide*. web page: http://www.microchip.com.

[23] G. Wang, "Execution time analysis on the PICos18 real-time operating system," Master's thesis, The Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2007.

[24] "Optimizing memory through automatically choosing communication protocols," *in preparation for publication*.