High Speed Hardware-Assisted Real-Time Interprocess Communication for Embedded Microcontrollers[†]

Sujaya Srinivasan

Intel Corporation
1900 Prairie City Rd. MS-FM26
Folsom CA 95630
sujaya.srinivasan@intel.com

David B. Stewart

Embedded Research Solutions, LLC.
9687F Gerwig Lane
Columbia, MD 21046
dstewart@embedded-zone.com

We have designed a hardware-assisted interprocessor communication (IPC) mechanism that can reduce communication overhead on embedded microcontrollers by a factor of thirty or more. The real-time communication mechanism allows processes to exchange data in a predictable and timely manner, with minimum overhead, in both single and multiprocessor environments. The hardware assist is a modified DMA architecture. We combined it with an existing software-based real-time communication mechanism, that takes advantage of the fact that in control systems, IPC requirements for most processes in the system are small and known a priori. The mechanism is designed especially for digital control systems that can be defined as a collection of software components that follow the port automaton model, for which it is more important to always obtain the most recent data, rather than always needing to process every data item.

1. Introduction

The design choices made by software engineers are often constrained by hardware limitations, as mechanisms that solve a problem in theory are not usable in practice, because they encounter too much overhead. For example, implementations of complex scheduling strategies, patterns and frameworks for object-oriented design, and interprocess communication can all require a significant amount of CPU time. On faster processors, the execution time might be negligible. Many embedded systems, however, must still use low cost and low power processors, for which complex strategies that may be suitable in theory cannot be implemented in practice.

In our software implementation of a variety of embedded applications, we have found many mechanisms that can provide the functionality we need, but we were unable to use the mechanisms on smaller microcontrollers because they dominate CPU time. One specific example is interprocess communication. While a handful of preemptive multitasking kernels have achieved reasonable context switch times on even the smallest 8-bit processors (like Nucleus for the MC6805), the presence of preemption introduces the need for interprocess communication and appropriate mutual exclusion for shared data. Implementation of mechanisms such as the priority ceiling protocol are not efficient, thus the ability to create a preemptive application is limited by the cost of providing interprocess communication.

The problems of IPC overhead become more substantial with the advent of systems-on-a-chip. In these cases, a single system might contain multiple 8-bit or 16-bit cores to provide coarse-level parallelism. A suitable multiprocessor IPC mechanism is necessary.

To address this issue, the most common form of mutual exclusion on embedded processors is to disable interrupts, often for extended periods of time. Doing so can adversely affect the real-time scheduling of processes, can lead to priority inversion, and may result in hard real-time tasks missing deadlines. Disabling interrupts is also not useful in multiprocessor environments, as it does not lock out processes executing on other processors. Other hardware mechanisms, like a test-and-set instruction, are more commonly used in multiprocessor applications, but they also incur a large overhead through busy waiting if the lock is not available.

An alternate solution is to strike a balance between efficiency and complexity, and select an IPC mechanism that does the job with some overhead, but hopefully not too much overhead. Taking this approach, we had developed the state variable (SVAR) communication mechanism, a black-board like architecture for exchanging data between processes [3,16]. While the mechanism uses less overhead than mechanisms such as message passing or semaphores, the overhead is still significant, and sometimes dominating, when using smaller embedded microcontrollers.

The objective of the work presented in this paper is to investigate the possibility of modifying the computer architecture in a way that can significantly reduce the overhead of this IPC mechanism, without introducing additional complexity for the software designer.

A hardware researcher is often hesitant to perform any architecture modifications that are custom to a specific application, as they are more interested in general solutions. The IPC mechanism that we selected, however, has been integrated into a real-time operating system (RTOS). In that regards, any application that uses the RTOS would in theory make use of the architecture modifications, and at the same time

[†] The reported research was performed at University of Maryland, where both authors were researchers prior to publication of this paper. The research was funded in part by NSF Division of Experimental and Integrated Activities, as Award No. 9806645, and in part by the Dept. of Electrical and Computer Engineering, University of Maryland.

abstract the hardware by keeping the same software interface that we are already using.

As software designers, the purpose of our work (and this paper) is to convince hardware designers to consider modifications to their traditional hardware designs, to better respond to the needs of our real-time software. The solution we propose combines the SVAR mechanism with a modified DMA mechanism to provide high-speed synchronized access to shared data. Our analysis shows that hardware assist can make IPC thirty times faster for a typical embedded control system! With these results, as presented here, we have motivated the computer architecture group at University of Maryland to begin architectural simulations of embedded applications with the proposed hardware modifications.

In Section 2, we review general mechanisms for providing IPC in real-time systems. In Section 3, we describe the SVAR mechanism that we have been using for IPC, and analytically compute the overhead for the SVAR mechanism. In Section 4, we present our design of a modified DMA mechanism that can be used to implement the most time-consuming parts of the SVAR mechanism in hardware. A performance analysis of the modified DMA mechanism is given in Section 5. In Section 6, we compare the use of a software-only SVAR vs. a hardware-assisted SVAR, to show the substantial reduction in overhead that results from IPC. Finally, in Section 7, we summarize our work, and provide an overview of the new computer architecture research that has begun to provide detailed simulations for a wider variety of embedded applications.

2. Background

The SVAR mechanism was selected as an IPC mechanism for implementing control systems that comprised of multiple components. The design of a control system using software components is described in [16]. The software decomposition is achieved by decoupling the functional computation from the interprocess communication. For example, each software component is implemented as a port automaton [15], with a basic structure of the code is as shown in Figure 1.

```
Initialize process
start = current time
begin loop
    x = readInputs( )
    Compute y = f(x)
    writeOutputs(y)
    start = start + period
    pause until next start time
end loop
```

Figure 1: General structure of a process following the port-automaton model

The premise of the port automaton model is that only the most recent data is of importance at any given time. It has been proven that a stable feedback control system can be achieved if each software component is implemented as an independent concurrent process [10], with the exception of communication through the input and output ports. An independent process executes without synchronization with other processes, thus each process may be execute at a different rate. When a process needs information, it obtains the most recent data available from its ***input ports***. From the process' standpoint, there is no knowledge as to the origin of the data obtained from this port; rather, the most recent data is sitting there, waiting to be read. When a process generates new information that might be needed by other processes, it sends this information to its ***output ports***. There is no knowledge as to who might look at this information placed on the output ports and there is no synchronization with other processes.

The port-automaton model was demonstrated experimentally in a robotics application, showing that stable control of a robot is indeed achieved [9]. The port-automaton model itself is not discussed further; details can be found in [16]. This model can be used in any system that is state-based, which includes the majority of sensor-based embedded systems. The rest of this paper describes the communication between software components that are designed using this basic port-automaton model.

A challenge to implementing a control system as a collection of concurrent processes is providing the link between outputs and inputs, while maintaining data integrity, guaranteeing real-time requirements, and managing the limited memory and CPU resources available on embedded microcontrollers. The requirements of the real-time communication mechanism necessary to support processes modeled as port automata are the following:

Data integrity: There must be no race conditions. Proper mutual exclusion or synchronization is needed to maintain data integrity. This ensures that one process does not read the data on its input port while another is changing it on the output port.

Asynchronous communication: The mechanism must be able to support communication between processes executing at different rates with minimal overhead. Many control systems like robotic systems require some processes to run at frequencies as high as 1000 Hz, while other processes may run much slower such as 30 to 100 Hz.

Low overhead: The communication mechanism should have low enough overhead to allow its implementation on embedded microcontrollers. Most embedded microcontrollers have only 8-bit or 16-bit data paths, have clocks speeds as low as 4 or 8 MHz, and have very limited memory, usually on the order of kilobytes.

Data broadcasting: The communication mechanism should be capable of fanning a single output to several inputs.

Traditional real-time communication mechanisms do not meet the above requirements. For example, message passing is a commonly used mechanism in real-time systems. Message passing is usually not a good solution for real-time systems with state data for the following reasons [16]:

- The most recent data is not always available, for example, if the producer produces data much faster than the consumer reads it, messages are queued, and the data used by the consumer may not be the most recent one.
- Message passing usually implies that a process is aware of the destination of its output, or the source of its input.
- In order to send an output to multiple processes, messages might need to be duplicated. Duplicating messages based

on number of recipients implies that the sending process is aware of the number of recipients, which violates the port-automaton model.
- The overhead of sending messages is too high to support processes that run at frequencies as high as 1000 Hz.

As a result of these problems with message passing, we instead consider communication mechanisms based on shared memory.

Shared memory communication in its most basic form is found to have much lower overhead than that associated with message passing. However, synchronization mechanisms are necessary to ensure that data integrity is maintained. In some cases, the overhead of synchronization is higher than that associated with message passing. The synchronization mechanism is needed to ensure that there are no race conditions. Some synchronization mechanisms may also have the potential for deadlocks; in a real-time system, deadlocks must be avoided at all costs, because rebooting the system is usually not acceptable.

On a uniprocessor, semaphores are used to guard access to shared memory, but they cause undesired blocking and priority inversion. The priority ceiling protocol [14] can be implemented to solve the problem of priority inversion, and it also provides a bounded blocking time for a process. However, the implementation of the priority ceiling protocol is complicated, and adds an unacceptable overhead for small embedded systems.

For multiprocessor applications, the *shared memory protocol* (SMP) [12] has been proposed. It is an extension of the single processor priority ceiling protocol. The protocol involves defining global semaphores for locking the global shared memory, and placing priority ceilings on accessing the semaphores to bound the waiting time of higher priority jobs. The overhead of implementing SMP is even greater than for the priority ceiling protocol, such that it is not suitable for many embedded systems. The SMP has a further limitation that it assumes all processes on all the processors are scheduled according to the rate monotonic algorithm [8] which may not necessarily be the case.

3. State Variable Table Mechanism

Our solution for providing real-time IPC has been to use the SVAR mechanism. The SVAR mechanism uses a two-level shared memory structure as shown in Figure 2. A global table in shared memory stores the state of all the variables in the system. In addition, every process maintains its own local copies of the data that it requires, and accesses data only from its local table during its cycle. Local tables are necessary in order to avoid race conditions. Just before a process executes its cycle, input data is copied from the global to the local table. The process then executes its cycle using the local table data. At the end of the cycle, output data is copied back from the local to global table. Since each process has its own local table, there are no possible race conditions while accessing this table. The only issue is maintaining integrity of the global table.

In a multiprocessor system, data integrity is achieved by using spin-locks on a global lock. When a process wants to access the global table, it locks out all other processes on its own CPU by disabling interrupts. This ensures that it does not get swapped out when it acquires the spin lock. The process then tries to acquire the global lock, by performing a *read-modify-write* instruction, which is supported in hardware by many processors. If it acquires the lock, it either reads or writes the global table, and releases the global lock. It then releases the lock on its own processor. If a process is unable to acquire the global lock, it spins on it, while still locking out other processes on its own processor. Spinning on the lock is not as critical as it seems, because it is guaranteed that the process holding the lock is on a different processor, and will release the lock shortly. To ensure that there are no deadlock conditions, a time-out can be generated to abort the spinning and release the global lock.

In a uniprocessor environment, synchronization is achieved by locking out other processes on the CPU, again assuming that the amount of data to be transferred is small [3]. In theory, this leads to priority inversion, and may cause tasks to miss deadlines. In practice, however, the priority inversion is less than the worst case operating system overhead and therefore, the reduction in predictability of the system is negligible. The operating system overhead on most embedded processors is of the order of 100μsec, which is the time to perform a system call such as a full context switch [3], or scheduling or handling a timer interrupt. Further, many RTOS are created such that periods and deadlines of processes are rounded to the nearest multiple of the system clock.

It is important to note that the global table is locked for only as long as it takes to transfer data between the global and local tables. If other mechanisms for accessing the shared data were used, such as the priority ceiling protocol, the global table (or parts of it) could be locked for the entire duration of the process.

3.1 Overhead of the SVAR Mechanism

The overhead associated with SVAR, though lower than message passing and SMP, becomes significant as the number of processes and the number of processors increase.

The overhead comprises two parts: (1) the data transfer time, and (2) the waiting time to acquire the lock. As the number of processes increases, the waiting time increases, and with a

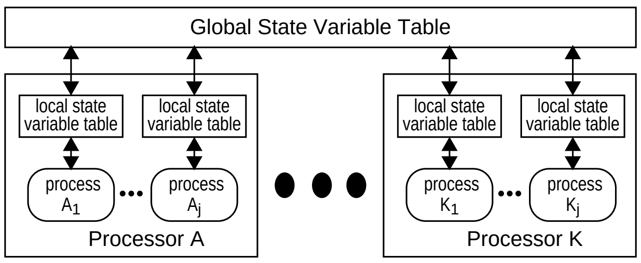

Global State Variable Table
local state variable table
local state variable table
process A_1
process A_j
Processor A
local state variable table
local state variable table
process K_1
process K_j
Processor K

Figure 2: Structure of state variable table mechanism for processes based on the port-automaton model

Table 1: Breakdown of transfer times and communication overhead for a Motorola 68030 25 MHz 32-bit processor (from [16])

Operation	Execution Time (μsec)
obtaining global state variable table lock using TAS	5
releasing global state variable table lock	2
locking CPU	8
releasing CPU lock	8
initial subroutine call overhead	4
block copy subroutine call overhead	7
total overhead for single variable read/write	34
additional overhead/variable for multivariable copy	5
raw data transfer over VMEbus, 6 floats	9
raw data transfer over VMEbus, 32 floats	31
raw data transfer over VMEbus, 256 floats	237

large number of processes, the waiting time becomes unacceptable. The analysis for computation of the data transfer and waiting times in a multiprocessor environment is given in [16] and the equations are repeated here. Similar analysis for a single-processor environment is in [3].

The analysis assumes that data is transferred from the global to the local table once at the beginning of the process' cycle, and output data is transferred once from the local to global table at the end of its cycle. If t_{IP} is the time to transfer the input variables and t_{OP} the time to transfer output variables of a process P, these values are given by

$$t_{IP} = V_1 + n_{IP}V_a + \sum_{i=1}^{n_{IP}} R(x_{Pi})$$

$$t_{OP} = V_{\phi 1}\phi + n_{OP}V_a + \sum_{i=1}^{n_{OP}} R(y_{Pi}) \qquad (1)$$

where V_1 is the overhead to lock and unlock the table; V_a is the additional overhead of transferring each variable; n_{IP}/n_{OP} are the number of input/output variables for the process P; x_{Pi}/x_{Po} are the number of transfers required for the *INVAR/OUTVAR* i of object P; and $R(x)$ is the time required for x transfers.

The overhead for data transfer times on a 25MHz Motorola MC68030 over a VME bus are shown in Table 1 (from [16]). The measurements obtained assume that there is no contention for the global lock. The locking and unlocking of the CPU are done through system calls to the RTOS. The raw data transfer times are non-linear, due to the underlying block copy routine, which have a better average time per byte for larger data transfers.

Any task on processor k attempting to lock the global table must wait for tasks on all higher priority processors. Furthermore, the task may also have to wait for a task currently holding the lock on a lower priority processor. Only one task on a processor can request the lock at once, and therefore, a process does not have to wait for processes on its own processor. The worst case waiting time for a process to acquire the global lock depends on the time that one lower priority process holds the lock, and the sum of the times that higher priority processes hold the lock. The worst case waiting time, W_k, for a process k, is given by

$$W_k = \sum_{j=1}^{k-1} \sum_{i=1}^{N_j} (t_{I,ij} + t_{O,ij}) + \max(M_j\big|_{j=k+1}^{r}) \qquad (2)$$

where $t_{I,ij}$ and $t_{O,ij}$ are the input and output data transfer times, M_j is the maximum time that a process j holds the lock. The first term in (2) corresponds to the waiting time for higher priority processes, and the second term is the waiting time for lower priority processes. The equation assumes a small volume of data transfer, where the worst-case of doing all transfers in sequence is less than the period of the fastest task. This calculation is based on the assumption that the volume of data is small and there is no starvation of lowest priority transfers. While this might not always be the case, it is a valid assumption for most embedded control applications.

An example of applying these equations to estimate the execution time of tasks when considering IPC overhead is given in Section 6. Next, we investigate the use of hardware assists to reduce the overhead and to speedup data transfers.

4. Hardware-Assisted IPC

The SVAR mechanism provides a relatively good software programming model for real-time communication between components. However, the overhead incurred by the software implementation of the SVAR communication—though much lower than that obtained using semaphores and message passing—is still significant, and is a limiting factor that prevents effective use of multitasking software on smaller embedded microcontrollers.

The overhead is primarily a result of two items:

- Data transfer time, which is the time spent in transferring data between the global and local tables. The data transfer time depends on the amount of data transferred into and out of each component.
- The waiting time to acquire the global lock. The waiting time increases as a function of the data transfer time and the number of processes.

We investigated the use of existing hardware to improve memory transfers. Direct memory access (DMA) was most promising, but for our needs the overhead incurred in setting up the DMA controller over-shadowed the gains in transfer time. Nevertheless, the use of DMA shows promise for improving the SVAR communication. In this section, we review existing DMA controllers, then show how a straightforward extension of existing technology can be used to greatly reduce the overhead of implementing the SVAR mechanism.

4.1 Traditional DMA Controllers

Hardware support in the form of DMA can reduce the data transfer time to copy data from one part of memory to another. DMA is a technique by which blocks of data can be transferred from external to internal memory, or from peripherals to main internal memory, without the intervention of the CPU.

In traditional DMA controllers, the parameters of a data transfer, namely the source and destination addresses and byte count, are set up in the DMA registers every time a DMA transfer is required. The overhead of setting up DMA transfers in this way is sometimes higher than the time to transfer a few bytes of data. Therefore, DMA is not used to transfer a few bytes of data. For larger blocks of data, the setup time becomes negligible compared to the data transfer time. Since most SVAR communication in a control system is for small amounts of data, using this type of DMA does not really improve the overall performance.

Traditional DMA controllers also do not provide the atomicity of transfers needed by the SVAR mechanism. For example, a cruise control task may need both position and velocity. Assuming that position and velocity are not stored contiguously in memory, the DMA needs to be setup separately for each of transfer; but in between transferring the position and setting up for velocity, the operations could get preempted by a higher priority task wanting to also perform DMA transfers.

Nevertheless, the concept of DMA to speedup transfers is appealing, especially if the overhead to setup transfers is reduced and the atomicity of transactions is addressed. We next look at an Enhanced DMA architecture that has addressed the issue of setup overhead.

4.2 Enhanced DMA

The Enhanced DMA (EDMA) controller in the newest TMS320C6211/6711 digital signal processor (DSP) provides the ability to link DMA transfers and 16 channels with programmable priority [18].

An event is a synchronization signal that triggers an EDMA channel to start a data transfer. The signal is a low-to-high transition on one of the 16 event input pins. Events can be individually disabled or enabled. An event can also be signalled by the CPU by writing a 1 to a bit in one of the EDMA registers.

The EDMA controller has a parameter RAM that stores the parameters for a particular EDMA transfer. In contrast, most DMA architectures are register-based, which makes it necessary to program the DMA registers every time a transfer has to be performed. Parameter entries can be linked to one another to provide for processing of complex streams or circular buffering.

The EDMA controller solves one of the problems with traditional DMA, since it does not require the process to set up the registers every time. However, linked EDMA transfers can be interrupted, and so they are not guaranteed to be atomic.

We propose a new DMA architecture, based on the EDMA, such that transfers can be pre-programmed, but the atomicity issue is also addressed.

4.3 Pre-programmed DMA (PDMA) Mechanism

The EDMA architecture allows parameters of data transfers to be stored in memory. Data transfers can be triggered by software or hardware at appropriate times. This solves the problem of traditional DMA controllers where the DMA registers have to programmed before every data transfer. However, it does not provide atomicity for input/output data transfers.

The proposed communication mechanism solves both the problems of traditional DMA controllers. We use the regularity of the port-automaton model to pre-program the DMA transfers, such that DMA transfers are done at pre-determined times between pre-determined memory locations. This mechanism, that we call Pre-programmed DMA (PDMA), is based on the SVAR communication mechanism, but enhanced with DMA hardware to perform data transfers. The hardware architecture is similar to the EDMA controller in the TMS320C6x digital signal processors. However, while the EDMA event can be triggered either in hardware by an external interrupt, which is a common application of EDMA since it is generally used to service peripherals, our PDMA is triggered only in software. Since the EDMA controller can interleave data transfers, an EDMA transfer is not necessarily atomic, whereas a PDMA transfer is guaranteed to be atomic.

DMA transfers can be pre-programmed if the port-automaton model described in Section 2 is used. During initialization of the system, a task pre-programs all of its input ports as transfers from global memory to local memory, and all of its output ports as transfers from local to global memory. By doing so, the PDMA mechanism does not require setting up the DMA registers before every data transfer. If a task has multiple input or output ports, these ports are linked, and the PDMA will guarantee that immediately upon terminating transfer of one variable, it will transfer the others.

Race conditions in the system are avoided since all transfers to and from shared memory are performed by the PDMA controller, and none of the processes can access the shared memory directly. Thus the necessity for a global lock is also eliminated.

Deadlock conditions are eliminated since the process is not directly involved in performing a data transfer. A process requests a data transfer (or the RTOS requests a data transfer on behalf of the process), then the PDMA controller takes care of the actual data transfer, and there are no resources locked up by the requesting process. Since PDMA transfers are guaranteed to be atomic, a process need not lock out other processes in order to wait for a data transfer to complete.

Using the PDMA provides DMA-speeds for the data transfer and eliminates most of the overhead by pre-programming transfers and guaranteeing atomicity of multiple inputs or outputs. In the next section, we discuss an implementation of the PDMA to show that it is feasible. A performance analysis is given in Section 5, and a comparison to the software-only SVAR mechanism is in Section 6.

4.4 Architecture of Pre-Programmed DMA

A system with multiple processes executing on a single processor is shown in Figure 3. Each of the processes has its own local table and all transfers between the local and global tables take place through the PDMA controller. Both the local and global memory can reside either on-chip or off-chip. On-chip memory has the advantage that memory access time is much faster than off-chip memory with wait states.

On a multiprocessor system, there are two design alternatives. First, the processors and the PDMA controller can be connected through a bus, over which only control information and parameters are transmitted. Data is not transmitted over this bus. The PDMA controller has another bus that gives it access to each of the local memories of the processors as well as the global shared memory as shown in Figure 4. The second alternative is to have all the PDMA registers memory-mapped, so as to enable all the processors to write control information to the registers through a memory write operation.

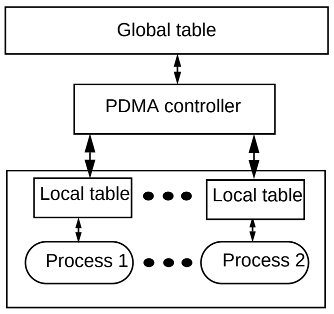


Figure 3: Flow of data between the process' local table and the global table through the PDMA controller

Each of the processors' local memories are dual-ported. Both the PDMA controller as well as the corresponding processor can access the memory in parallel, as long as they do not try to access the same memory location. In case of a contention, the PDMA controller is given priority, and the memory accesses are serialized. If the memory is not dual-ported, then the processor cannot access its local memory when PDMA transfers are taking place. The architecture is suitable for both PC-board and system-on-a-chip implementations.

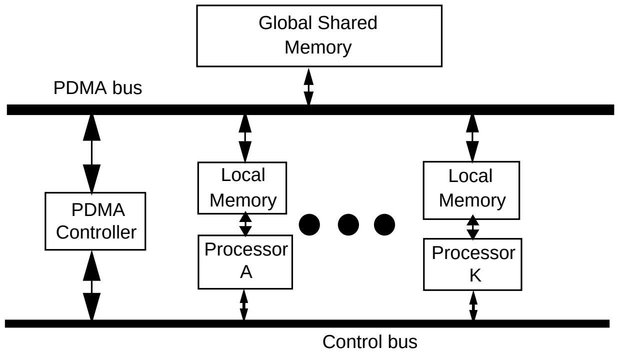


Figure 4: Multiprocessor environment, where each processor has multiple processes executing on it. PDMA controller performs all transfers between local and global memory

The PDMA controller must be designed such that it has registers and a memory to store entries for data transfers to be performed. Information about the data transfers is programmed at system startup.

A process has zero or more input variables that are transferred from the global table to the process' local table at the beginning of its cycle, and zero or more output variables that are written to the global table from the process' local table, at the end of its cycle. It is not possible to guarantee that input variables (or output variables) are in contiguous memory locations in the local and global tables. For example, in Figure 5, process 1 has input variables, x_2 and x_3, which are contiguous in the global table and output variables x_4 and x_6, which are non-contiguous in the global table. A variable can be a vector, in which case each element of the vector is in contiguous memory locations in both the global table and local tables. To pre-program the PDMA, the source and destination addresses and byte count are stored for each variable and for each process. The parameters for a single variable transfer then constitutes an entry in the PDMA memory. If a process has zero input or output variables, then the PDMA controller is not programmed for that transfer.

A PDMA transfer is a chain of data transfers constituting the input variables or output variables of a process. For example, if a process has three input variables, each having different source and destination addresses, the PDMA transfer consists of a chain of three consecutive DMA transfers corresponding to each of the variables. A PDMA transfer is atomic, meaning that the PDMA controller completes the transfer for each of the input variables or the output variables of a process, before it starts on the next PDMA transfer. This implies that a single PDMA transfer may consist of several entries, which are linked to each other. Details are explained in the next section.

4.4.1 PDMA registers and memory

The PDMA controller needs the following registers:

- PDMA enable register
- Trigger register
- Link pointer register
- Source address register
- Destination address register
- Byte count register

The PDMA enable register has a bit that needs to be set for the PDMA controller to be enabled. This is necessary in order

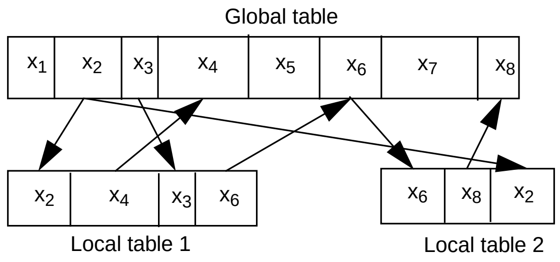


Figure 5: Data in local and global tables. Arrows indicate the transfers that need to be pre-programmed.

to disable the PDMA controller if new entries need to be reprogrammed. The PDMA controller only checks the enable bit when it completes a transfer, it cannot be disabled during a transfer. When the PDMA controller finds the enable bit reset, it acknowledges that it is disabled by setting the acknowledge bit.

The trigger register has one bit corresponding to each PDMA transfer. The bit is activated when the corresponding PDMA transfer is ready to take place. The maximum number of PDMA transfers is determined by the size of this register, because each bit in the trigger register directly maps to a PDMA entry. So, if the PDMA register is 32 bits wide, there can be a maximum of 16 processes in the system, if each process has both input and output transfers. For smaller microcontrollers that only have a few kilobytes of memory, it is unusual to exceed 16 processes. For larger microcontrollers, multiple PDMA registers can be used to provide as many bits as necessary.

The link pointer register has the address of the entry for the next data transfer that is to take place. In the case of PDMA transfers that involve more than one entry, the link pointer register has the address of the next entry corresponding to the same transfer. It is necessary in order to link data transfers.

The source address and destination address register have the source address and destination address for the DMA transfer. During a DMA transfer, they are incremented every read or write cycle, until the byte count becomes equal to zero. The byte count register contains the number of bytes to be transferred.

The PDMA controller has entries for the data transfers stored in memory, as shown in Figure 6. As stated earlier, there may be more than one entry corresponding to each PDMA transfer. Each entry, which corresponds to one input or output variable, has the source address, destination address and byte count for that variable. There is also a field containing the address of the next entry corresponding to the PDMA transfer. An address of NULL in the next entry field means that there are no more data transfers corresponding to the particular PDMA transfer.

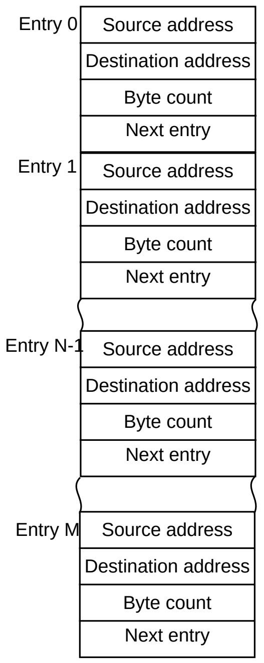


Figure 6: PDMA controller memory showing data transfer entries.

4.4.2 Priorities of data transfers

Priorities are assigned to PDMA transfers, such that the PDMA controller always performs higher priority ready transfers before lower priority ones. In the simplest case, a PDMA transfer with a lower entry number has lower priority than one with a higher PDMA entry number. This is a static priority scheme. In more complex implementations, PDMA transfers can be assigned priorities through a priority register, allowing assignment of priorities dynamically. But, the cost of this implementation is that the PDMA controller needs more time to find the highest priority ready PDMA transfer. For example, if there are 32 PDMA transfers, the PDMA controller has to check the each of the bits in the transfer in the order of priority, and this would, in the worst case take 32 cycles. This affects the setup time for a PDMA transfer. The simpler implementation is likely sufficient for most applications, because the time taken for PDMA transfers is small, and therefore a high setup time is unjustified.

If the full bandwidth of the PDMA is used, then the static priority assignment can lead to starvation for lower priority transfers. In control applications, however, this is rare, as the amount of data that is transferred between processes is usually small as compared to the computation time used by the process. Whereas the goal of using DMA is often throughput, that is not the case for PDMA. Rather, the goal of PDMA is to minimize the computational resources used for IPC, to free up those resources for the critical control code.

4.4.3 PDMA operation

The operation of the PDMA controller is shown in Figure 7. The PDMA controller is always enabled except when the entries are to be reprogrammed. When the PDMA controller is not in the process of transferring data, it looks for the first 1 in the trigger register. When it finds a 1, it finds the corresponding entry in memory and loads the source, destination, byte count and link pointer registers. For example, if a process requests PDMA transfer 2, it sets bit 2 of the trigger register. The PDMA controller then loads the values in entry 2 from memory into its registers and starts performing the transfer. When it finishes performing the data transfer, it looks for the entry in its memory corresponding to the address in the link pointer register. If the address in the link pointer register is NULL, then it

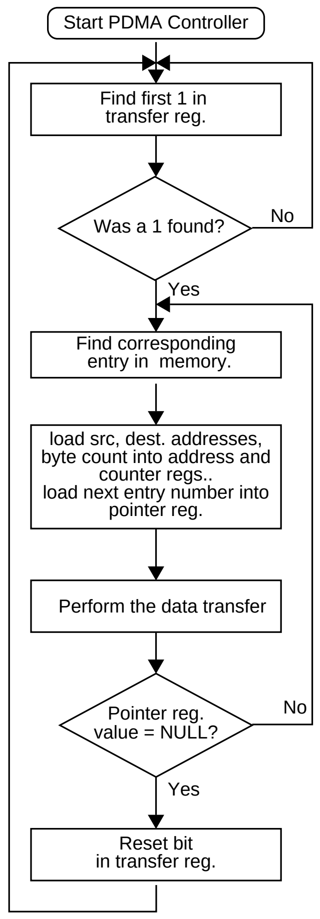

Start PDMA Controller
Find first 1 in transfer reg.
Was a 1 found?
No
Yes
Find corresponding entry in memory.
load src, dest. addresses, byte count into address and counter regs.. load next entry number into pointer reg.
Perform the data transfer
Pointer reg. value = NULL?
No
Yes
Reset bit in transfer reg.

Figure 7: Flowchart showing operation of the PDMA controller

means that the PDMA transfer is complete. The PDMA controller resets the corresponding bit in the trigger register, and continues checking for other ready entries.

From a software perspective, performing the data transfer is then simply a matter of setting the bit in the trigger register to 1. A PDMA transfer is guaranteed to be atomic, since the PDMA controller completes the entire chain of entries before it checks the trigger register for any other ready transfers. A process may either wait for the transfer bit to be reset, or the operating system could perform other tasks which do not require access to shared memory. The process may even be preempted, since there are no deadlocks or race conditions that arise due to its preemption.

4.5 Software Interface to PDMA

A key aspect of the PDMA is that it is designed for use by the RTOS, not the application programmer. The application-programmer interface (API) of the SVAR mechanism is used to abstract the hardware details. The steps followed by a process using the PDMA mechanism are shown in Figure 8, and incorporated into the RTOS. When a process is scheduled, the RTOS sets the appropriate bit in the trigger register (single cycle operation), to request the input data transfer. It then waits on the completion of the data transfer. The worst case waiting time is computed in Chapter 5. At the end of a process' execution, the RTOS again sets the bit in the trigger register (single cycle) corresponding to the output data transfer for the process. The process does not need to wait until the output transfer is done, because it has finished computing its output. The computed output data is then written back to shared memory, making it available for other processes which need it.

Once the processes are initialized, all the data transfers to and from shared memory are transparently performed by the RTOS when the application calls *ReadInputPorts()* or *WriteOutputPorts()* (refer to Figure 1). This API is consistent with the software-only version of the SVAR mechanism. Thus an application that uses SVAR, but does not have PDMA hardware present, can revert to the software implementation, without the need to change the application code.

```
Create local table
Specify input variables and output variables
Program entries into PDMA table
start = current time
begin loop
    set bit in trigger register for input data transfer
    wait until trigger register bit reset
    Compute y = f(x)
    set bit in trigger register for output transfer
    start += period;
    pause until next start time
end loop
```

Figure 8: Process using the PDMA mechanism. The steps shown in *bold italics* are operations performed by the RTOS

4.5.1 PDMA reconfiguration

The PDMA mechanism makes use of the fact that most of the processes' information is available at initialization, which allows pre-programming of data transfers. However, some processes may be created or destroyed at runtime. The PDMA mechanism allows entries to be programmed dynamically.

The PDMA controller is disabled during reconfiguration. The RTOS resets the enable bit in the PDMA enable register. If the PDMA controller is in the middle of a transfer, it completes it before being disabled, and sets the acknowledge bit in the PDMA enable register. The RTOS writes the appropriate entries into the PDMA memory, and then enables the PDMA controller again.

When a new process is created dynamically, entries for the process need to be programmed into the PDMA memory. The RTOS first reads the entry number of the first free entry. By this scheme, the new data transfer is given the highest available priority. It then writes the values of source address, destination address, byte count and next entry into the PDMA memory. This procedure takes approximately 10 cycles. This means that to program 5 entries, the PDMA controller is disabled for about 50 clock cycles, which is approximately 6µsec on an 8MHz processor or 1µsec on a 50MHz processor. Since reprogramming is not a common occurrence, the overhead is negligible.

When a process is destroyed dynamically, and its entries are no longer in use, the RTOS adds these entries into the free list of entries. The PDMA controller need not be disabled during this time, since the entries need not be actually erased from the PDMA memory. The PDMA entries can be reused once they are on the free list. Since this operation does not involve the PDMA controller itself, it is not a critical overhead.

5. Performance Analysis of the PDMA Mechanism

The overhead associated with the PDMA mechanism comprises two parts: (1) the data transfer time and (2) the waiting time before the PDMA controller starts the requested data transfer. The data transfer time is the time taken by the PDMA controller to perform the requested data transfer, including the setup time. The waiting time is the interval between the time that a process requests a data transfer (by setting the appropriate transfer bit) to the time that the PDMA controller starts the requested data transfer. A process waits for the data transfer to complete by busy-waiting on the transfer bit. While added execution time can be saved by performing useful work while awaiting the transfer to complete, we do not take such savings into account; but this does imply the potential for improvement beyond what we describe here.

5.1 Data transfer time

Data transfer time consists of the time to perform the actual data transfer, and the system overhead to perform the data transfer. The time to transfer the data depends on the memory architecture of the system. If all memory is on-chip, then the PDMA controller needs one cycle to read data from the

Table 2: Breakdown of communication overhead for data transfers with zero wait states (times in μsec)

Operation	Clock cycles	Time on an 8MHz 16-bit bus	Time on a 25MHz 32-bit bus
scan transfer register to find first 1	1	0.125	0.040
write transfer register bit on completion	1	0.125	0.040
system overhead incurred only once	2	0.250	0.080
find appropriate entry in PDMA memory	1	0.125	0.080
load pointer, address and counter regs.	4	0.500	0.160
overhead for each variable	5	0.625	0.240
transfer 4 bytes of data		0.500	0.080
transfer 24 bytes of data		3.000	0.480

source, and one cycle to write data to the destination. There may be additional wait states if the memory is off-chip.

If c_r is the number of cycles (including wait states) to read data from the source addresses, c_w is the number of cycles (including wait states) to write data to the destination address, B_w is the bus width of the PDMA controller bus in bytes, and f_{PDMA} is the frequency of the PDMA controller bus in MHz, then the throughput of the PDMA controller, t in bytes/msec. is given by

$$\tau = (f_{PDMA} * B_w) / (c_r + c_w) \qquad (3)$$

Note that this is the worst-case throughput. In the case where there are wait states for both read and write operations, the wait states may overlap and the actual throughput may be higher than the calculated t. On a PDMA controller with a 32-bit wide 25MHz data bus and zero wait state memory where c_r and c_w are one cycle each, the throughput of the PDMA controller, τ is 50Mbytes/sec.

The time taken to scan the transfer register at the beginning and write the transfer register bit on completion is the overhead incurred only once during each PDMA transfer and is denoted by t_0. Scanning the transfer register can be performed by an instruction such as the Find First One (FF1), which is available on many Motorola processors. For example, in the Motorola MCORE [11], this operation takes one clock cycle. Simple hardware similar to the find-first-one unit on the MCORE can be built to make this a one-cycle operation. At the end of the PDMA transfer, the PDMA controller sets the transfer bit to indicate completion.

Each PDMA transfer may consist of a number of linked data transfers, and each time, the transfer parameters are loaded from the PDMA memory into the address, link pointer and counter registers. This overhead is denoted by t_a.

The number of clock cycles for each of these operations, and the actual times for data transfer on a PDMA controller with an 8MHz 16-bit bus and one with a 25MHz 32-bit wide bus with no wait states is shown in Table 2. The number of clock cycles is multiplied by the cycle time to get the actual times.

Let t_j be the transfer time for data transfer j. Let n_j be the number of variables for transfer j, and x_i be the number of bytes for each variable. Then t_j is computed as

$$t_j = t_0 + n_j t_a + \frac{1}{\tau} \sum_{i=1}^{n_j} x_i . \qquad (4)$$

The transfer calculations are used as a basis for evaluating the projected performance of the PDMA if it is implemented as an architectural extension.

6. Performance Comparison

In this section, a comparison of the software-only SVAR mechanism with the PDMA-Assisted SVAR is performed.

An example of a control system is shown in Figure 9. Each box is a process that is designed using the framework that was shown in Figure 1. Each arrow represents an SVAR. If the arrow enters a box, it is an input variable. If it exits a box, it is an output variable. For comparison purposes, we will assume that the size of each variable is the same.

In Table 3 through Table 5, we compare the software-only and PDMA-assisted mechanisms for three different configurations:

- 25 MHz 32-bit microcontroller, each variable is 24 bytes.
- 8 MHz 16-bit microcontroller, each variable is 24 bytes.
- 8 MHz 16-bit microcontroller, each variable is 4 bytes.

In the tables, W is the worst case waiting time as calculated by Eqn (2). The priorities of the processes are assigned based on the inverse of their period, i.e, the process with the smallest period has the highest priority.

Note that only the overhead values for the software-only mechanism are measured values on real processors and they are consistent with the values obtained from equations. The rest of the values are computed using the equations in this paper. The task set is hypothetical, with periods and execution times chosen to show utilization at approximately 90%.

Among these three cases, the improvements range from 27 to 46 times faster for the PDMA-assisted mechanism. The improvement for each task is also shown graphically in Figure 10. Each bar shows the percentage of overhead, computed as a ratio between the execution time of the IPC mechanism and the base execution time of the task.

By comparing the first two cases, we see that there is a significant improvement for both the faster and larger 32-bit pro-

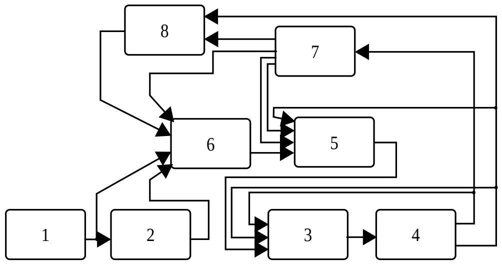


Figure 9: Example of a multitasking control system.

Table 3: Comparison of SVAR mechanisms, 25 MHz 32-bit processor, with 24-byte data transfers

			Software SVAR					SVAR Using PDMA					
PID	T Period msec	C WCET μsec	t_{in} μsec	t_{out} μsec	W Waiting Time μsec	modified WCET μsec	δ_s % over-head for SVAR	t_{in} μsec	t_{out} μsec	W Waiting Time μsec	modified WCET μsec	δ_p % over-head for PDMA	Improve-ment δ_s/δ_p
1	40	1000	34	48	448	530	53.0	0	1	11	1012	1.1	45.7
2	10	2000	48	48	352	2448	22.4	1	1	9	2011	0.5	41.6
3	4	1000	76	48	90	1214	21.4	2	1	3	1006	0.5	37.7
4	100	2000	48	62	668	2778	38.9	1	1	15	2017	0.8	44.8
5	8	2000	90	48	214	2352	17.6	3	1	6	2009	0.4	38.1
6	50	3000	90	48	530	3668	22.2	3	1	12	3015	0.5	44.1
7	400	5000	48	90	798	5936	18.7	1	3	17	5020	0.4	46.1
8	200	4000	62	48	778	4888	22.2	1	1	17	4020	0.4	45.4

Table 4: Comparison of SVAR mechanisms, 8 MHz 16-bit processor, with 24-byte data transfers

			Software SVAR					SVAR Using PDMA					
PID	T Period msec	C WCET μsec	t_{in} μsec	t_{out} μsec	W Waiting Time μsec	modified WCET μsec	δ_s % over-head for SVAR	t_{in} μsec	t_{out} μsec	W Waiting Time μsec	modified WCET μsec	δ_p % over-head for PDMA	Improve-ment δ_s/δ_p
1	120	3000	19	114	1558	4691	56.3	0	4	56	3060	2.0	28.2
2	30	6000	114	114	1330	7558	25.9	4	4	48	6056	0.9	28.6
3	12	3000	304	114	399	3817	27.2	11	4	15	3030	1.0	27.2
4	300	6000	114	209	2204	8527	42.1	4	8	79	6090	1.5	28.1
5	24	6000	399	114	817	7330	22.1	15	4	30	6048	0.8	27.7
6	150	9000	399	114	1691	11204	24.4	15	4	60	9079	0.9	27.8
7	1200	15000	114	399	2451	17964	19.7	4	15	87	15106	0.7	27.8
8	600	12000	209	114	2527	14850	23.7	8	4	90	12102	0.9	27.9

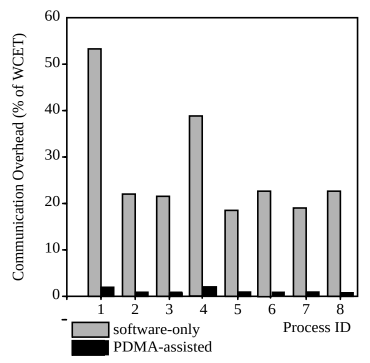
Table 5: Comparison of SVAR mechanisms, 8 MHz 16-bit processor, with 4-byte data transfers

			Software SVAR					SVAR Using PDMA					
PID	T Period msec	C WCET μsec	t_{in} μsec	t_{out} μsec	W Waiting Time μsec	modified WCET μsec	δ_s % over-head for SVAR	t_{in} μsec	t_{out} μsec	W Waiting Time μsec	modified WCET μsec	δ_p % over-head for PDMAμsec	Improve-ment δ_s/δ_p
1	80	3000	19	43	493	3555	18.5	0	1	15	3016	0.5	34.9
2	20	4000	43	43	407	4493	10.9	1	1	13	4015	0.4	32.8
3	8	2000	91	43	115	2249	11.0	3	1	4	2008	0.4	31.9
4	200	6000	43	67	713	6823	13.7	1	2	21	6024	0.4	34.3
5	16	4000	115	43	249	4407	9.2	4	1	8	4013	0.3	30.8
6	100	9000	115	43	555	9713	7.9	4	1	16	9021	0.2	34.4
7	800	15000	43	115	818	15976	6.5	1	4	24	15029	0.2	34.3
8	400	12000	67	43	823	12933	7.9	2	1	24	12027	0.2	33.8

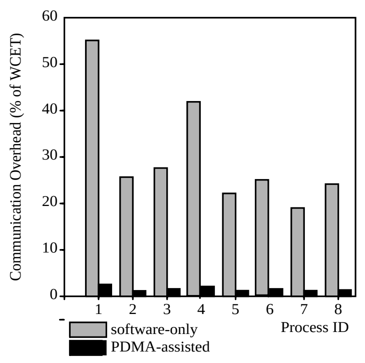
cessor, and the smaller and slower 16-bit processor. In the third graph we see that even if the data transfers are small (i.e. 4 bytes), using the PDMA can still provide significant improvement. In contrast, traditional DMA devices are usually only effective for larger transfers.

7. Summary and Future Work

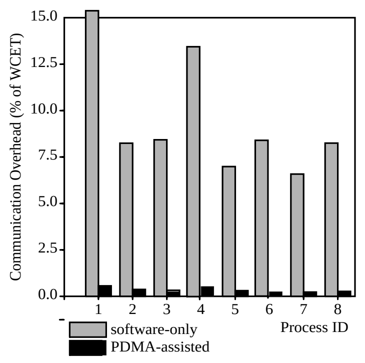
The three example cases represent an extremely small sampling of possibilities. Because the values we use for comparison are primarily computed and not measured, extending the number of sample sets that we look at would not prove the PDMA approach to be any better or worse. Rather, what is needed is a detailed architectural simulation of the PDMA, with a variety of embedded applications executed as benchmarks. As a software group, these types of simulations are beyond our expertise. Rather, the results that we present say, "As software designers, this is the way we would like the



(A) 25 MHz, 32-BIT CPU, 24-BYTES DATA



(B) 8 MHz, 16-BIT CPU, 24-BYTES DATA



(C) 8 MHz, 16-BIT CPU, 4-BYTES DATA

Figure 10: Graphical comparison of IPC overhead for software-only and PDMA-assisted SVAR mechanisms.

DMA hardware to be built." This information has been passed along to a computer architecture group at the university, who is now in the process of extending their MCORE simulator to include PDMA transfers. Considering many architectural enhancements are designed to improve performance of key code by perhaps 30 or 40 percent, the possibility of building new hardware that can improve performance 30 times (i.e. 3000 percent) is very appealing. We expect results of those simulations to be published at a later time.

For a computer architect, it is not desirable to build hardware mechanisms that are not sufficiently general for a large number of applications. Because the proposed mechanism can be encapsulated by an RTOS, any application that uses the RTOS can thus use the new hardware; thus the PDMA has the potential to not only be excellent at reducing IPC overhead, but it can do so for a large number of applications, in particular any application that uses state data. There might also be other applications for PDMA beyond the domain of embedded real-time systems, in which case PDMA might be suitable for general-purpose processors too.

8. References

[1] T. Cantrell, "Flash Forward," *Circuit Cellar Ink*, March 1999.

[2] R. C. Dorf, Modern Control Systems, 3rd Ed., Addison Wesley, 1980.

[3] M. Hassani, David. B. Stewart, "A Mechanism for Communicating in Dynamically Reconfigurable Embedded Systems," in *Proc. of High Assurance Software Engineering Workshop*, Washington DC, August 1997.

[4] R.Grehan, "Eight-bit processors - All over the Map,", *Embedded Systems Programming*, February 1999.

[5] D. Hildebrand, "Message-passing Operating Systems," *Dr.Dobbs Journal*, June 1998.

[6] Intel Corporation, "i960 RM/RN I/O processor developer's manual".

[7] B. L. Jacob, "Cache design for embedded real-time systems," *Embedded Systems Conference*, Danvers, MA, June 1999.

[8] C. L. Liu, and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment," *Journal of the ACM*, v.20, n.1, pp. 44-61, January 1973.

[9] D. M. Lyons and M. A. Arbib, "A formal model of computation for sensory-based robotics," *IEEE Transactions on Robotics and Automation*, v.5, n.3, pp. 280-293, June 1989.

[10]L. D. Molesky, C. Shen, G. Zlokapa, "Predictable synchronization mechanisms for multiprocessor real-time systems," *The Journal of Real-Time Systems*, v.2, n.3, September 1990.

[11]Motorola,"MCORE Reference Manual MCORERM/AD".

[12]R. Rajkumar, "Real-Time Synchronization Protocols for Shared Memory Multiprocessors," in *Proc. of 10th International Conference on Distributed Computing Systems*, Paris, France, pp. 116-123, June 1990.

[13]Ready Systems, "Ready Systems VRTXvelocity System Programmer's Guide, Volume VI".

[14]L. Sha, R. Rajkumar, J.P. Lehoczky, "Priority Inheritance protocols: an approach to real-time synchronization", *IEEE Transactions on Computers*, vol.39, no.9, p.1175 - 85, Sept. 1990.

[15]M. Steenstrup, M. A. Arbib, and E. G. Manes, "Port automata and the algebra of concurrent processes," *J. of Computer and System Sciences*, v.27, n.1, pp. 29-50, Aug. 1983.

[16]D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Trans. on Software Engineering*, v.23, n.12, Dec. 1997.

[17]D. B. Stewart, G. Arora, "Dynamically Reconfigurable Embedded Software - Does it make sense," *IEEE Intl. Conf. on Engineering of Complex Computer Systems and Real-Time Applications Workshop*, Montreal, Canada, Oct. 1996.

[18]Texas Instruments,"TMS320C6000 Peripherals Reference Guide".

[19]H. Tokuda, T. Nakajima, "Evaluation of Real-Time synchronization in Real-Time Mach," *USENIX 1991 Mach Workshop*, October 1991.