# Atalanta:
# A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications

Di-Shi Sun
Douglas M. Blough
Vincent John Mooney III

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250, USA

## Abstract

This paper introduces a new multiprocessor real-time operating system (RTOS) kernel that is designed as a software platform for System-On-Chip (SoC) applications and hardware/software codesign research purposes. This multiprocessor RTOS kernel has the key features of an RTOS, such as multitasking capabilities, event-driven priority-based preemptive scheduling; and interprocess communication and synchronization. Atalanta has some features important for SoC applications, such as a small, compact, deterministic, modular and library-based architecture. Atalanta also supports some special features such as priority inheritance, and user configurability. Atalanta supports multiple processors with synchronization based on atomic read-modify-write operations.

## Keywords

Real-Time Operating System (RTOS), System-on-Chip (SoC), Hardware/Software Co-design, Multiprocessor architecture

# 1. Motivation

Hardware/software codesign research needs a solid software platform to evaluate hardware and software ideas for SoC applications. There are several commercial RTOSes for SoC applications, but only a few support multiprocessor architectures. Multiprocessor architectures have many variants making the RTOS very complicated if general multiprocessor architectures must be supported. Hence, our project focuses on a particular target architecture that is commonly used in SoC. The target architecture is described in Section 4.

Because we want to implement the software platform on different processor targets and support heterogeneous multiprocessor architectures, it is necessary to use a compiler and binary tools that can support a variety of processor families. For this reason, we use GNU cross-compilers and binutils in our research, and we use CVS as our source code control tool.

# 2. Goals

The new RTOS should have key features of an RTOS. These features include the following:

- Multitasking capabilities
- Event-driven, priority-based, preemptive scheduling
- Interprocess communication and synchronization

The RTOS should also have some important features for SoC applications. These features include the following:

- Small and compact
- Deterministic
- Modular and library-based architecture

The RTOS should also support some additional special features such as the following:

- Priority inheritance
- High level of user configurability
- I/O architecture

The most important feature Atalanta should support is the following:

- Homogeneous and heterogeneous multiprocessors on the target hardware architecture described in Section 4.

We want the RTOS to be easy to use, easy to port and possibly to support different processor families. The source code should be clean, consistent and well organized. The higher-level operating system functions such as the user interface, file system, and network support are not included in the current release of the kernel.

# 3. Development Status

Homogeneous multiprocessor versions of Atalanta for PowerPC and ARM debugged under Seamless CVE™ have been released. This version of Atalanta is being evaluated by multiple hardware/software co-design research projects with both ARM and PowerPC processors.

A version of Atalanta for heterogeneous multiprocessor architectures is in the planning stage. A hardware description is currently being developed in order to test the heterogeneous OS. Additional topics under investigation are the design of an I/O system for Atalanta and methods to optimally configure Atalanta for a given application.

# 4. Target Hardware Architecture

The hardware architecture for Atalanta is a tightly coupled SoC architecture. Multiple processors, memory and other components are placed into a single chip. These processors may be from different processor families and be used for different purposes. They reserve some of the memory for exclusive by a single processor and share the remainder for storing global data structures, and performing interprocessor communication and synchronization. Heterogeneous multiprocessor architectures with differing processor types are the topic of a separate RTOS development effort in our group.

We assume there is no other hardware connection between processors except shared memory over a bus. A hardware timer interrupt is required to generate a system clock signal for Atalanta. We assume that each processor has its own timer interrupt. We assume there are atomic read-modify-write instructions in the instruction sets of our target processor families. We need the read-modified-write instruction in order to implement spin lock.

# 5. The Architecture of Atalanta

Atalanta is an RTOS kernel; this means Atalanta only provides the fundamental functions of an RTOS. Some higher-level RTOS functions such as user interface and network support are not provided.

## 5.1 Software architecture and execution model

For homogeneous multiprocessor architectures, in Atalanta, all of the processors share the same address space. Only one copy of the Atalanta kernel and the application reside in memory. Each processor in the system uses the same program and shares all of the data structures.

The application and Atalanta kernel are compiled together as a single program. Each processor loads the same program but runs it as a separate process. Different processors can execute a different sets of tasks based on the processor identification (PID). Multiple real-time tasks can be included in each process. This architecture has obvious advantages. It can save memory, but the most important is it is easy to implement inter-processors communication and synchronization. For heterogeneous multiprocessor architectures, our separate development effort uses shared-memory-based message passing to implement interprocessor communication and synchronization.

## 5.2 Spin-lock

We use hardware atomic read-modify-write operations to implement spin lock in Atalanta. In order to simplify the system design, we just use one system spin lock. So the whole kernel looks like a critical section. This means that when a task calls a system call, the tasks running on other processors cannot make any system call until the first system call is finished. This is not a

problem for Atalanta, because every system call in Atalanta is extremely short in duration and Atalanta only supports up to 16 processors.

### 5.3 Layered architecture

An SoC application based on Atalanta is layered in accordance with function. The lowest level is the kernel level. The kernel level contains the simplest operating system routines, such as scheduling routines and list operation routines. All hardware dependent routines are in this level. The next level is the system level. It contains all of the system object management routines. The top level is the application level. It contains user-defined, project dependent application code.

In Atalanta, each level should only make use of the routines provided by the level below. Application level code should not directly call the routines provided by kernel level. For example, it is strongly recommended that application code not call enable/disable interrupt routines. Interrupt Service Routines (ISRs) are treated like application level routines in Atalanta, i.e. ISR should only call system level routines but should not call kernel level functions.

### 5.4 Static architecture

Atalanta is designed to allocate memory for every system object statically. So the memory requirement of an application is determined at compile time. This is a very important feature for SoC applications. All of the system objects are automatically created at initialization time according to the configuration of the application and cannot be deleted at run time.

### 5.5 Modular and library-based architecture

The Atalanta kernel is designed as a modular and library-based architecture. The kernel and all of the system object management routines are placed in a single library. An application can link the functions that are used from the library automatically.

## 6. Task Management

### 6.1 Static task assignment

Atalanta uses a static task assignment policy. Every task is assigned a processor on which to run at initialization time. Tasks cannot run on other processors. In fact, it is easy to move a task running on one processor to another processor in homogeneous multiprocessor environment (if an application needs this feature).

### 6.2 Task scheduling

Atalanta uses the same scheduling scheme on every processor in homogeneous multiprocessor environment. Atalanta schedules tasks directly based on each task's static priority. In general, the task with higher priority can get control of processor earlier than the task with lower priority. In a group of equal-priority tasks, either First-In-First-Out (FIFO) order or round robin scheme may be used. Calling a system service routine may cause an immediate preemptive rescheduling if the system routine makes the calling task no longer the highest priority ready task.

## 6.3 Tasks in Atalanta

Atalanta supports up to 255 tasks –including idle tasks and device driver tasks. Every task must be assigned a priority between 0 and 127. Usually, the lowest priority 127 is assigned for idle tasks and highest priority 0 is assigned to device driver tasks. Atalanta does not support dynamic task creation or deletion. Tasks are automatically created only during system initialization time. To exit from a task's startup routine will cause a call to asc_task_suspend for the task. The kernel does not automatically release the kernel resources that this task holds; the task must release these resources before it exits. It will cause an unpredictable result to resume this task again by calling asc_task_resume routine.

In Atalanta, a task has four basic states: ready, blocked, waiting, and suspended. Initially, all tasks are in the ready state. When a task is waiting for a system object without a timeout value, it is placed in the blocked state. When a task is sleeping or waiting for a system object with a timeout value, it is placed in the waiting state. When a task is suspended, it is in the suspended state. That means a task can be in several states or combined states as follow: ready; blocked; waiting; suspended; waiting and suspended; blocked, waiting and suspended.

The priority of tasks can be modified at run time. This is a kernel feature. An application cannot use this feature directly; it is only used by our priority inheritance mechanism, which is implemented in the kernel.

## 6.4 Task usage

Tasks can reference the status of, suspend, or resume a task. Tasks also can disable/enable the scheduler on any processor.

## 6.5 Task management system calls

There are seven task management routines in Atalanta.

- *asc_task_sleep*
  This system call is used to move the calling task to the ready state or the waiting state. If the time-out parameter is non-zero, the calling task is placed in the waiting state. If a zero timeout value is specified, processor control is given to another ready task with equal or higher priority, and the calling task is immediately put back in the ready queue.

- *asc_task_suspend*
  This system call is used to move a task, either the calling task or another, to the suspended state. A call to asc_task_resume must be made to resume the task.

- *asc_task_resume*
  This system call Is used to remove another task from the suspended state. If the task is also in the blocked state and/or waiting state, this routine only clears the suspended state but leaves the task in the blocked state and/or waiting state. The asc_task_suspend and asc_task_resume should be used as a pair.

- *asc_task_wakeup*
  This system call Is used to wake up another task that is in either blocked state, or waiting state or both. If the task is not also suspended, it will be moved to the ready state.

- *asc_task_reference*
  This system call Is used to obtain information about a task; for example the task ID, the processor ID on which the task is running,the task priority level, and the task state.

- *asc_lock*
  This system call Is used to disable the preemptive rescheduler on a single processor.

- *asc_unlock*
  This system call is used to re-enable the preemptive rescheduler on a single processor. The lock/unlock routines are only effective on a single processor.

# 7. Interprocessors/Interprocesses Communication and Synchronization

Atalanta provides several system objects to implement interprocessors/interprocesses communication and synchronization. Tasks and ISRs can pass pointer-size messages using mailboxes and queues or use event groups for synchronization. Semaphores and mutexes can be used to control access to resources.

Based on the target hardware architecture, Atalanta uses shared memory to implement interprocessor communication and synchronization for a homogeneous multiprocessor architecture. For a heterogeneous multiprocessor architecture, Atalanta will combine shared memory with message passing to implement interprocessor communication and synchronization.

All of the system objects used for interprocess communication and synchronization can also be used for interprocessor communication and synchronization. A difference can be seen when a task wakes up another task on another processor and places the other task in the ready state; Atalanta just places the task into the ready queue, but cannot make the task run immediately. The task must wait for at most one time tick. Hardware interrupts between processors could be used to cause the scheduling to occur immediately but this is beyond our target hardware configuration.

## 7.1 Event group management

### 7.1.1 Event groups in Atalanta

An event group is a set of up to 32 event binary flags in Atalanta. When a flag is set, this means that an event has occurred. Atalanta supports up to 255 event groups, but does not support dynamic event group creation or deletion. Event groups are automatically created only during system initialization.

### 7.1.2 Event group usage

An event flag can be used by a task or ISR to inform another task of the occurrence of an event. A task can wait for an OR condition on multiple event flags. When any one of the flags is set, the task is placed in the ready state. Also, a task can wait for an AND condition on multiple event flags. The task is not placed in the ready state until all of the event flags have been set at the same time. This means if a task is waiting for two events, if one event occurs and is cleared and then another event occurs, the task would not be placed in the ready state.

More than one task can wait for an event at the same time. This means a task or ISR can broadcast the event to all of the tasks that are waiting for it.

### 7.1.3 Event group management system calls

There are five event group management routines in Atalanta.

• *asc_event_clear*
This system call Is used to clear one or more event flags in a specified event group.

• *asc_event_signal*
This system call Is used to set one or more event flags in a specified event group. Every task that is satisfied by these flags will be placed in the ready state if the task is not also in the suspended state.

• *asc_event_gain*
This system call Is used to poll for one or more event flags (AND or OR) condition in a specified event group. The caller is not blocked (placed in the blocked state) if the event flags do not satisfy the caller's condition.

• *asc_event_seek*
This system call is used to wait for one or more event flags (AND or OR) condition in a specified event group with time-out. The caller would be blocked (placed in the blocked state with/without waiting state) if the event flags do not satisfy the caller's condition. The caller remains blocked until the condition is satisfied, a time-out occurs or the caller is woken up by another task or ISR.

• *asc_event_reference*
This system call is used to obtain current event flags states in a specified event group.

## 7.2 Mailbox management

### 7.2.1 Mailboxes in Atalanta

A mailbox is a one pointer-size buffer in Atalanta. It supports up to 255 mailboxes. Atalanta does not support dynamic mailbox creation or deletion. Mailboxes are automatically created only during system initialization.

### 7.2.2 Mailbox usage

Mailboxes allow a task or ISR to pass pointer-size nonzero messages to another task. More than one task can wait on a mailbox at same time. This means when a message is sent to that mailbox, the highest-priority task receives the message and is placed in the ready state.

### 7.2.3 Mailbox management system calls

There are five mailbox management routines in Atalanta.

• *asc_mailbox_post*
This system call is used to send a message to a mailbox. The calling task is not blocked if there is already a message in the mailbox. Instead, it returns immediately with an error code. If there is any task waiting on the mailbox, *asc_mailbox_post* removes the first task from the wait queue and places it in the ready queue if it is not also suspended.

• *asc_mailbox_send*
This system call Is used to send a message to a mailbox with a time-out value. The calling task is blocked if there is already a message in the mailbox. The task remains blocked until either the message is removed from the mailbox, the time-out period elapses, or the caller is woken by another task or ISR. If there is any task waiting on the mailbox, the routine removes the first such task from the wait queue and places it in the ready queue if it is not also suspended.

• *asc_mailbox_gain*
This system call is used to receive a message from a mailbox. The calling task is not blocked if the mailbox is empty. Instead, it returns immediately with an error code.

• *asc_mailbox_seek*
This system call is used to receive a message from a mailbox with a time-out value. The calling task is blocked if the mailbox is empty and it remains blocked until either a message is sent to the mailbox, the time-out period elapses, or the caller is waked by another task or ISR.

• *asc_mailbox_reference*
This system call Is used to retrieve a message from a mailbox without removing it from the mailbox.

## 7.3 Queue management

### 7.3.1 Queues in Atalanta

A queue is a fixed-length pointer-sized message buffer in Atalanta. It supports up to 255 queues; one queue can have up to 255 pointer-sized messages in it. Atalanta does not support dynamic queue creation or deletion. Queues are automatically created only during system initialization.

### 7.3.2 Queue usage

Queues allow a task or ISR to pass pointer-size messages to another task. Messages can be

handled in First-In-First-Out (FIFO) or Last-In-Fist-Out (LIFO) order in a queue. A queue with a length of one is identical to a mailbox.

### 7.3.3 Queue management system calls

There are five queue management routines in Atalanta.

- *asc_queue_post*
  This system call is used to send a message to a queue. The calling task is not blocked if the queue is full. Instead, it returns immediately with an error code. If there is any task waiting on the queue, the routine removes the first such task from the wait queue and places it in the ready state if it is not also suspended.

- *asc_queue_send*
  This system call is used to send a message to a queue with a time-out value. The calling task is blocked if the queue is full. It remains blocked until either a message is removed from the queue, the time-out period elapses, or the caller is woken by another task or ISR. If there is any task waiting on the queue, the routine removes the first such task from the wait queue and places it in the ready queue if it is not also suspended.

- *asc_queue_gain*
  This system call is used to receive a message from a queue. The calling task is not blocked if the queue is empty. Instead, it returns immediately with an error code.

- *asc_queue_seek*
  This system call is used to receive a message from a queue with a time-out value. The calling task is blocked if the queue is empty. It remains blocked until either a message is sent to the queue, the time-out period elapses, or the caller is waked by another task or ISR.

- *asc_queue_reference*
  This system call is used to retrieve information about the state of a queue, namely the contents of the first message and the total number of messages in the queue. The state of the queue is not modified by the call.

## 7.4 Semaphore management

### 7.4.1 Semaphores in Atalanta

In Atalanta A semaphore is a counter variable used for synchronization. Atalanta can support up to 255 semaphores and every semaphore can have up to 255 resources. Atalanta does not support dynamic semaphore creation or deletion. Semaphores are automatically created only during system initialization.

### 7.4.2 Semaphore usage

In Atalanta, semaphores can be used as both binary and counting semaphores. However, semaphores are normally used as counting semaphores and mutexes are used as binary

semaphores (mutexes supports priority inheritance). A counting semaphore can be used to control access to a set of two or more application resources. Semaphores should not be used to control access to kernel resources, which are protected by spin locks.

Using nested semaphores can cause deadlock. Atalanta does nothing to avoid, prevent, or recover from deadlock. Deadlock avoidance and/or handling are the responsibility of the application.

### 7.4.3 Semaphore management system calls

There are four semaphore management routines in Atalanta.

• *asc_semaphore_signal*
This system call is used to release a resource to a semaphore. The semaphore count is incremented by one. If there is any task waiting on the semaphore, the routine removes the first task from the wait queue and places it in the ready queue if it is not also suspended.

• *asc_semaphore_gain*
This system call is used to receive a resource from a semaphore. The calling task is not blocked if there is not any available semaphore. Instead, it returns immediately with an error code.

• *asc_semaphore_seek*
This system call is used to receive a resource from a semaphore with a time-out value. The calling task is blocked if there is not any resource in the semaphore. It remains blocked until either a resource is released to the semaphore, the time-out period elapses, or the caller is woken by another task or ISR.

• *asc_semaphore_reference*
This system call is used to obtain the resource count of a semaphore.

## 7.5 Mutex management

### 7.5.1 Mutexes in Atalanta

in Atalanta a mutex is a binary semaphore, which supports priority inheritance. Atalanta supports up to 255 mutexes. Atalanta does not support dynamic mutex creation or deletion. Mutexes are automatically created only during system initialization.

### 7.5.2 Mutex usage

A mutex can be used to control access to a single application resource, but normally a mutex is used to enforce mutual exclusive access to a critical section in an application. A task must obtain the mutex before it enters the critical section and the task must release the mutex before it leaves the critical section.

### 7.5.3 Mutex management system calls

There are four mutex management routines in Atalanta.

• *asc_mutex_signal*
This system call is used to release a mutex. If there is any task waiting on the mutex, the routine removes the first such task from the wait queue and places it in the ready queue if it is not also suspended.

• *asc_mutex_gain*
This system call Is used to acquire a mutex. The calling task is not blocked if the mutex is unavailable. Instead, it returns immediately with an error code.

• *asc_mutex_seek*
This system call is used to acquire a mutex with a time-out value. The calling task is blocked if the mutex is unavailable. It remains blocked until either the mutex is released; the time-out period elapses, or the calling task is waked by another task or ISR.

• *asc_mutex_reference*
This system call Is used to obtain the state of a mutex without modifying it.

# 8 Memory Management

Atalanta uses partitions to manage memory.

## 8.1 Partitions in Atalanta

A partition is a group of fixed-size memory blocks that can be dynamically allocated and released at run time in Atalanta. It supports up to 255 partitions. Every partition can have up to 255 blocks. Every block can have up to 64k bytes, the block number and size is fixed. Atalanta does not support dynamic partition creation or deletion. Partitions are automatically created only during system initialization.

## 8.2 Partition usage

Fixed-size memory blocks can be dynamically allocated from and released to the memory pool of a partition. This method supports the static architecture of Atalanta as well as keeps enough flexibility. Because mailboxes and queues can only pass pointer-size messages, use of a memory partition is the only way supported by Atalanta to pass large data structures of significant contents.

## 8.3 Partition management system calls

There are four partition management routines in Atalanta.

• *asc_partition_gain*
This system call is used to get a memory block from a partition. The calling task is not blocked if there is no free memory block in the partition. Instead, it returns immediately with an error code.

• *asc_partition_seek*
This system call is used to get a memory block from a partition with a time-out value. The calling task is blocked if there is no free memory block in the partition. It remains blocked until either a memory block is released to the partition, the time-out period elapses, or the calling task is waked by another task or ISR.

• *asc_partition_free*
This system call is used to release a memory block to a partition. If the memory block is not allocated from the partition, the result is unpredictable. The application must make sure to free memory blocks to their correct partitions. If there is any task waiting on the partition, the routine removes the first such task from the wait queue and places it in the ready queue if it is not also suspended.

• *asc_partition_reference*
This system call is used to obtain the status of a memory partition, namely the number of allocated memory blocks, the number of free memory blocks, and the size of each memory block in the partition.

# 9 Timer Management

Atalanta supports system clock and virtual timers.

## 9.1 System clock

### 9.1.1 System clock in Atalanta

Atalanta maintains a 32-bit system clock counter. At every clock tick, an interrupt handler increases the current system clock counters.

### 9.1.2 System clock usage

The system clock counter is provided for applications. Kernel routines do not use this counter.

### 9.1.3 System clock management system calls

There are two system clock management routines in Atalanta.

• *asc_gettime*
This system call is used to retrieve the system clock value measured in Atalanta clock ticks.

• *asc_settime*
This system call is used to set the system clock value.

## 9.2 Virtual timer

### 9.2.1 Virtual timer in Atalanta

Atalanta supports up to 255 “one-shot virtual” timers. When a virtual timer is started with a time-out value, the timer reduces its value by one at every clock tick. When the time-out value reaches zero, a user-defined action is performed. Atalanta does not support dynamic virtual timer creation or deletion. Virtual timers are automatically created only during system initialization time.

### 9.2.2 Virtual timer usage

Virtual timers can be used to perform a user-defined action after a certain time interval.

### 9.2.3 Virtual timer management system calls

There are four virtual timer management routines in Atalanta.

• *asc_timer_define*
This system call is used to specify an action for a virtual timer.

• *asc_timer_start*
This system call is used to specify a relative time and to start a virtual timer.

• *asc_timer_stop*
This system call is used to stop a virtual timer.

• *asc_timer_reference*
This system call is used to obtain the time remaining on a virtual timer.

# 10 Interrupt Management

Atalanta does not filter or redirect interrupts. Every hardware interrupt will stop the application and pass control to an ISR. Atalanta provides two kernel calls that can be used as a part of the ISR.

## 10.1 Interrupt stack switch

Atalanta supports interrupt stack switching. When the first nested interrupt occurs, Atalanta switches the stack from the interrupted task stack to a uniform interrupt stack. Subsequent nested interrupt ISRs will run on the uniform stack. When the last nested interrupt ISR exits, Atalanta switches the stack back to the interrupted task stack. The advantage of the interrupt stack switch is to reduce stack usage because the application does not have to allocate a large stack space for every task.

## 10.2 Interrupt in Atalanta

Atalanta supports nested interrupts. Atalanta provides support for interrupts nested up to 127 levels.

## 10.3 Interrupt management system calls

There are two interrupt management routines in Atalanta.

*aic_enter*
This call enters an ISR, increases the interrupt counter by one, and performs an interrupt stack switch if necessary.

*aic_exit*
This call exits an ISR, decreases interrupt counter by one, does interrupt stack switch back if necessary. It also may make a rescheduling call if necessary. In fact, the *aic_enter* and *aic_exit* are kernel calls rather than system calls. Atalanta places them in the ISR stub and an application cannot call them directly.

# 12 Future Work

Developing a variety of applications that use Atalanta is underway. Some of the features of Atalanta need to be adjusted according to the application. There are many features we would like to include in the future releases of Atalanta such as the following:

- Supporting the POSIX interface.
- Adding support for the Floating Point Unit (FPU) and the Memory Management Unit (MMU).
- Supporting user-defined callout functions.

Designing methods to optimally a configure Atalanta for a given application is an additional area of research. The support of heterogenous and multiprocessor architectures is one of features that will be supported in the next version of Atalanta.

# References

[1] Andrew S. Tanenbaum and Albert S. Woodhull, *Operating Systems: design and implementation 2nd Ed.*, Prentice Hall, 1997.

[2] Abraham Silberschatz and Peter Baer Galvin, *Operating system concepts 5th Ed.*, Addison Wesley Longman, Inc., August 1998.

[3] Jean J. Labrosse, *MicroC/OS-II The Real-Time Kernel*, R&D Books, October 1998.

[4] Lui Sha, Raghunathan Rajkumar and John Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transaction on Computers*, vol. 39 no 9, September 1990, pp. 1175-1184.

[5] H. Kopetz: "Event-triggered versus Time-triggered real-time systems," *Lecture notes in Computer Science*, Vol. 563. Springer Verlag, Berlin, 1991, pp. 87-101.

[6] Mentor Graphics, Hardware/Software Co-Verification: Seamless CVE™, http://www.mentor.com/seamless/.

[7] On-Line Applications Research Corporation, *Getting Started with RTEMS for C/C++ Users*, 1st Edition, May 2000.

[8] On-Line Applications Research Corporation, *RTEMS ITRON 3.0 User's Guide*, 1st Edition, September 2000,