Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2010, Article ID 601343, 15 pages
doi:10.1155/2010/601343

Research Article

An Embedded System Dedicated to Intervehicle Communication Applications

Xunxing Diao,[1] Haiying Zhou,[2] Kun-Mean Hou,[1] and Jian-Jin Li[1]

[1] *LIMOS Laboratory, UMR 6158 CNRS, Blaise Pascal University Clermont-Ferrand II, Aubière 63173, France*
[2] *School of Computer Science, Harbin Institute of Technology, Harbin 150001, China*

Correspondence should be addressed to Haiying Zhou, haiyingzhou@hit.edu.cn

Received 1 December 2009; Revised 30 March 2010; Accepted 7 July 2010

Academic Editor: Guoliang Xing

Copyright © 2010 Xunxing Diao et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

To overcome system latency and network delay is essential for intervehicle communication (IVC) applications such as hazard alarming and cooperative driving. This paper proposes a low-cost embedded software system dedicated to such applications. It consists of two basic component layers: an operating system, named HEROS (hybrid event-driven and real-time multitasking operating system), and a communication protocol, named CIVIC (Communication Inter Véhicule Intelligente et Coopérative). HEROS is originally designed for wireless sensor networks (WSNs). It contains a component-based resource-aware kernel and a low-latency tuple-based communication system. Moreover, it provides a configurable event-driven and/or real-time multitasking mechanism for various embedded applications. The CIVIC is an autoconfiguration cooperative IVC protocol. It merges proactive and reactive approaches to speed up and optimize location-based routing discovery with high-mobility nodes. Currently, this embedded system has been implemented and tested. The experiment results show that the new embedded system has low system latency and network delay under the principle of small resource consumption.

1. Introduction

Each year in Europe, 1,300,000 vehicle accidents result in 1,700,000 personal injuries. The financial cost of vehicle accidents is evaluated at 160 billion Euros (approximately the same cost in the USA [1]). Many IVC projects were investigated [2–4] but the implementation aspects were not detailed. To improve the highway safety, a low-cost and a more reliable embedded IVC is needed especially for applications like hazard alarming and cooperative driving (e.g., collision avoiding). As one can imagine, such applications require extra effort to deal with real-time event and network delay under the dynamic topology caused by highly mobile network nodes; thus, we have proposed an autoconfiguration location-based IVC protocol named CIVIC in [5, 6].

General purpose proactive (e.g., OLSR [7]) and reactive protocols (e.g., AODV [8]) are not adapted to IVC application due to high dynamic topology change. The CIVIC protocol includes both proactive and reactive approaches to make it suitable for IVC. The proactive approach is the one-hop neighbour knowledge exploration. In order to avoid network traffic overhead, the proactive intervals are autoconfigured depending on the positions and speeds of network nodes. Based on previous neighbour information, CIVIC can then speed up and optimise the routing discovery. The routing approach can be reactive or proactive depending on application layer requiring. Either way, if destination node is not in one-hop distance, CIVIC will select the best effective node to forward routing requests by a directional resource-aware broadcast mechanism.

The last experiment result of CIVIC protocol is shown in [5]. Until this experiment, the tasks in CIVIC are implemented as infinite loops. Tasks are driven by events (e.g., timer interrupt) and run in a nonpreemptive scheduling mechanism. Such mechanism cannot assure the event-driven tasks run in time when system is busy, and it is difficult to achieve the intranode resource-aware.

To overcome these shortcomings, this paper proposes a new low-memory footprint IVC design integrated an operating system named HEROS. HEROS merges the advantages from both event-driven and real-time multitasking

mechanisms into a hybrid configurable component-based mechanism. This hybrid mechanism can be adopted in various applications driven by events but also required to have real-time operations. For example, in the intervehicle hazard alarming applications, when a vehicle detects the hazard triggered by events, it may need to maintain real-time communications to inform other vehicles for example to avoid collision (in case of bad weather: smog, snow, etc.).

A fundamental requirement for practical embedded system is resource-aware, HEROS provides the new IVC embedded system with intranode resource-aware mechanism. Although the embedded system on vehicles may gain better hardware supports, the characteristics of embedded hardware still have to cope with resource constraints in terms of CPU, memory, energy, and transmission distance. The HEROS originally designed for WSNs with stringent resource constraints. Its microkernel architecture allows hybrid tasks to be run with low memory consumption. Moreover, it provides a tuple-based intranode communication and synchronization system based on the parallel programming language LINDA [9, 10]. It is the key technology to enable the lightweight resource-aware design on our embedded IVC system.

Summarizing, in the new IVC embedded system, HEROS provides CIVIC with intranode mechanisms to run hybrid tasks and manage hardware, while CIVIC constitutes a quick-response internode communication stack on HEROS. The designs have been implemented in LiveNode sensor board [11], and experimented with a small network grouped by nine nodes. The experiment results show the new design embedded system has low system latency and network delay. Thus it is adapted to IVC application such as collision avoidance.

The remainder of the paper is organized as following. The related works of HEROS and CIVIC will be summarized in the next section. Section 3 introduces the HEROS system microkernel. Section 4 presents the CIVIC protocol. Section 5 explains how these two component layers work together. Section 6 describes the evaluations of system performance. In the last section, we present the conclusion and the ongoing work.

2. Related Works

2.1. Embedded Operating System. In the existing embedded OSs, there are two common operation mechanisms: multitasking and event-driven.

The real-time multitasking mechanism provides a solution for rapidly developing the time-sensitive applications and it gives the full control over tasks [12]. However, this mechanism consumes high resources in terms of energy, CPU and memory. The existing embedded RTOSs such as SDREAM [13], μC/OS-II [14], VxWorks, QNX, pSOS, WinCE.NET, RTLinux, Lynxos, RTX, and HyperKernel are not suitable for the embedded IVC system because they only operate as this mechanism and it is resource consuming (CPU and memory) comparing with our proposed solution one.

To minimize resource consuming, many embedded OSs were developed for WSN fields (called WSNOS: WSN Operating System) such as TinyOS [15], MagnetOS [16], Contiki [17], MantisOS [18], EYEOS [19], and SOS [20] (sensor operating system). These WSNOSs meet the requirement of resource constraint. TinyOS adopts the event-driven component-based structure and has a tiny memory footprint. The rest of WSNOSs except Contiki adopt multitasking concept. Similar to TinyOS, Contiki is based on event-driven, but it may be configured to run in hybrid mode: event-driven and multitasking. Contiki is not a native hybrid WSNOS.

Note that, on one hand, a single task event-driven system does not fit for hard real-time constraint. On the other hand, in an event-driven mechanism (e.g., TinyOS), the task switches is normally based on a nonpreemptive event-loop. This mechanism has the advantage in low resource consumption, so it is suitable for WSNs. However, the existing event-driven embedded WSNOSs are essentially implemented by a single processing mechanism; thus, they may not be suitable for IVC applications, which require complex real-time operations.

In HEROS, we merge these two operation mechanisms into a configurable modular mechanism. This design is able to adapt to more various WSN and IVC applications include intelligent transportation, health care, military, and so forth.

2.2. IVC Protocol. The major features of CIVIC protocol are to discover and maintain the routing path in high-mobility embedded networks. The current routing protocols can be classified into three classes: proactive, reactive, and hybrid.

The proactive routing protocols maintain up-to-date routing tables for partial or entire network. It keeps the message delay low because data can be sent to a destination node without an immediate routing request. However, in order to have correct routing paths, each node needs to explore network routing periodically, thus network traffic could be increased significantly. The main proactive protocols are OLSR (optimized link state routing) [7], DSDV (destination-sequenced distance-vector) [21], and TBRPF (topology dissemination based on reverse-path forwarding) [22]. Moreover, the proactive routing protocols are not suitable for IVC because the network topology changes quickly due to the vehicle mobility.

The reactive routing protocols do not maintain routing tables. They discover routing paths only when a demand is received. Therefore, they are more efficient in terms of bandwidth utilisation but along with additional message delay. The main reactive protocols are DSR (dynamic source routing) [23], AODV (ad hoc on demand distance vector) [8], TORA (temporally-ordered routing algorithm) [24], ABR (associativity-based routing) [25], and SSR (Signal Stability Routing) [26].

The hybrid routing protocol combines the advantages of the proactive and reactive protocols. An example is ZRP (zone routing protocol) [27]. It includes two routing components: a proactive intrazone routing component and a reactive interzone routing component. The CIVIC protocol has some similarities like ZRP. However, none of previous

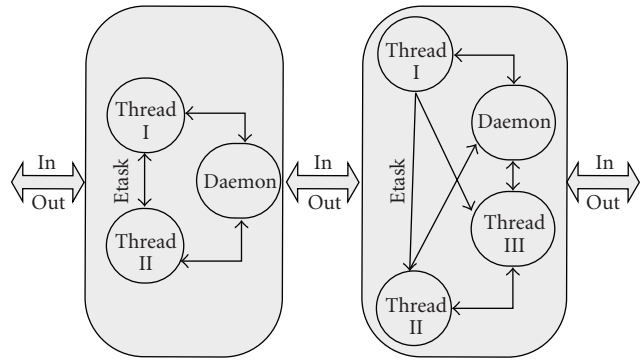


Figure 1: HEROS component-based architecture: thread and etask.

mentioned routing protocols have considered the particularity of the IVC such as direction, location, and road traffic, which will be explained in detail in Section 4.

3. HEROS Microkernel

The cost and the efficiency of the embedded IVC system are an important factor for car manufacturers and highway infrastructure management. To minimize the cost it is essential to implement appropriate embedded hardware and software (real-time operating system and communication protocol) meeting the real-time IVC application.

In this section, the system architecture, the scheduling mechanism, and the communication and synchronization mechanism of HEROS microkernel are introduced, respectively.

3.1. System Architecture. HEROS adopts the component-based system architecture to perform the event-driven and/or real-time operations. It contains two main system components: thread and etask (event task).

Thread is the essential system component that performs a single action in HEROS. A series of threads can be engaged in complex real-time task under the control of a master etask. Threads belong to an etask run parallel and corporately; hence, they must be interruptible and preemptive. Each etask must contain at least the general daemon thread, which enables the related hardware to be switched into low-power mode. For example, in an application with low wireless data rate, most of time, the daemon thread can disable the wireless access medium module (idle mode). Etask is a packing widget that encapsulates a group of threads to complete a task. Etasks are performed in sequence according to the priority of etasks; hence, etasks are interruptible but not preemptive. Within an etask, threads share tuple space (buffer resources) and allocate private context stacks. After an etask is completed, its tuple and stack will be released. This design allows the embedded application to be scheduled for various tasks with less memory footprint.

The communication of components (threads and etasks) is via a mutual tuple space. In the first time, an etask is activated, this etask generates a tuple space for its slave threads. The tuples will not be released when leaving an etask.

Table 1: Structure of component control block.

ID	Numeric ID of this component
STAT	Current state of this component
PRI	Priority of this component
MAX_TIME	Maximal lifetime of this component
CUR_TIME	Current runtime of this component
NXT_ITEM	Pointer of next component in the ready list
TUPLE_ID	Numeric ID of the thread's tuple
SSP	Start buffer pointer of the thread's stack
CSP	Current buffer pointer of the thread's stack

Threads and etasks calls the IN/OUT system primitives to exchange data and transfer message/signal via the relative tuple space. A thread is triggered by signals coming from other components or external peripherals. An etask is activated only after one of its threads is triggered by a signal. The component-based system architecture is shown in Figure 1.

In a HEROS implementation with only one event containing multiple threads, the threads can be scheduled in fully real-time multitasking mode. In an implementation with multiple events but each contains only one thread (besides daemon thread), the tasks can be run in event-driven mode like TinyOS. In software design, etask and thread components are represents as two data structures: etask control block (ECB) and thread control block (TCB) as shown Table 1.

3.2. System Scheduling. HEROS adopts the two-level priority-based scheduling mechanism to merge event-driven and real-time tasks at one system. This mechanism can provide a predictable scheduling with an invariable scheduling time.

3.2.1. Priority Scheduling Mechanism. Due to the interruptible and nonpreemptive characterises, etasks are performed in a typical event-driven mode. An active etask runs to completion until all threads of this etask has been terminated. In view of the interruptible and preemptive characteristics, threads are performed in a typical multitasking mode. The elected thread can preempt any other lower priority

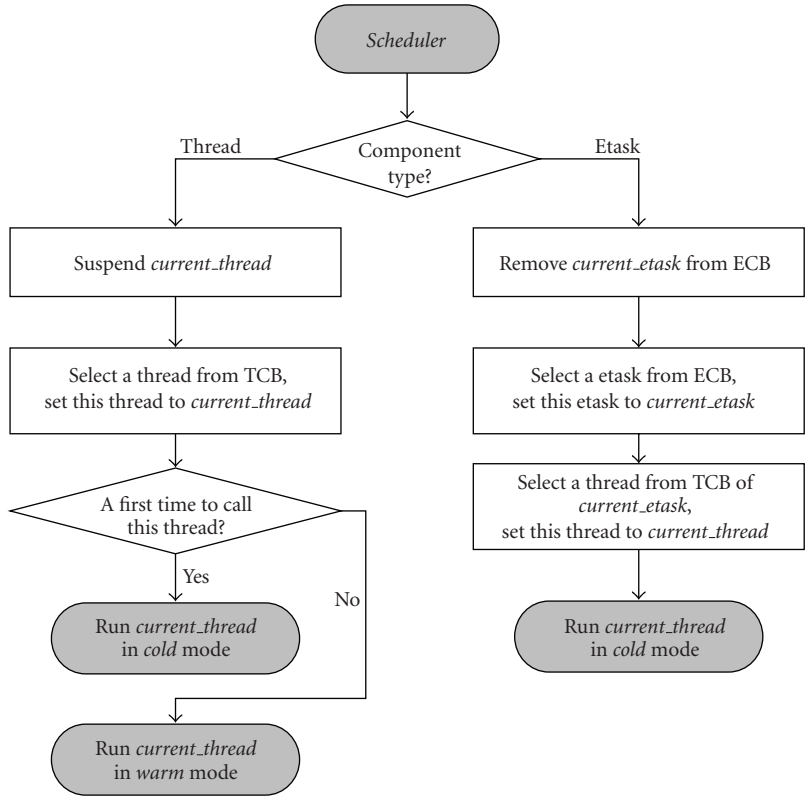


Figure 2: HEROS system scheduling mechanisms.

threads at any execution point outside of system critical section.

The priority of system components (etask and thread) are calculated as the following expression. Defining P_{cur} is the component priority, then

$$P_{\text{cur}}(t) = \left(1 + \frac{t}{T_{\max}}\right) \times P_{\text{cur}}(0), \quad 0 \leqslant t \leqslant T_{\max}, \tag{1}$$

where $P_{\text{cur}}(0)$ and $T_{\max}$ are the initialization constants and $P_{\text{cur}}(t)$ is a time function, $0 \leq t \leq T_{\max}$. $P_{\text{cur}}(0)$ and $T_{\max}$ indicate the initial component priority and the maximal allowable lifetime "MAX_TIME," which are preallocated when this component is activated at time 0 ($t = 0$). The "CUR_TIME" value T_{cur} can thus be expressed as follows:

$$T_{\text{cur}}(t) = T_{\max} - t, \quad 0 \leqslant t \leqslant T_{\max}, \tag{2}$$

where $T_{\text{cur}} = 0$ at time $T_{\max}$, which means that the component elapses its time-slice and then will be terminated.

Both etasks and threads adopt the "priority-based" scheduling mechanism, in which the etask scheduling is nonpreemptive and the thread scheduling is preemptive. The system-scheduling flowchart is shown in Figure 2 and the main scheduling functions are listed in Table 2. The execution time of scheduling functions is predictable and deterministic.

Table 2: System scheduling functions.

In	System Primitive, read data from tuple
Out	System Primitive, write data into tuple
Etask_Manager	Perform etask scheduling mechanism
Thread_Scheduler	Perform thread scheduling mechanism
InsertThreadList	Insert a thread to TCB and resort TCB items
DeleteThreadList	Delete a thread in TCB and resort TCB items
InsertEtaskList	Insert a etask to ECB and resort ECB items
DeleteEtaskList	Delete a etask in ECB and resort ECB items
EtasktoThread	Perform etask-to-thread conversions

3.2.2. Etask-to-Thread Conversion. Considering a specific case where a higher priority etask ε_H is ready and at the same time the current activated etask ε_L has the lower priority comparing with ε_H one, then ε_H can only be elected to run after ε_L has been terminated. Consequently, the above-mentioned scheduling mechanism cannot meet the real-time requirement.

One solution is to adopt the *etask-to-thread conversion* scheme: ε_H can be treated as the thread τ with highest priority in ε_L, so that τ_s can preempt any other active thread of ε_L in view of the thread scheduling mechanism. The scheme breaks down the obstacle between threads and etasks, allowing the threads of an urgent etask to preempt the CPU resource in real-time.

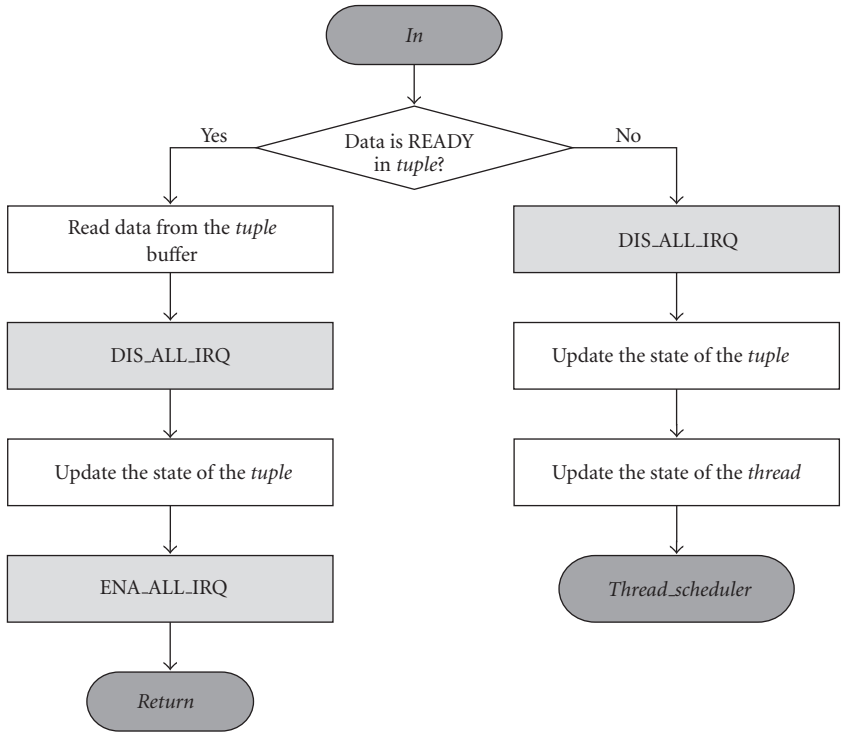


Figure 3: Functional descripticon of the IN system primitive.

Table 3: Structure of tuple_table.

ID	Numeric identifier: key of tuple
STAT	Current tuple state: Free or Full
WRI_HEAD	Current writing buffer pointer
REA_TAIL	Current reading buffer pointer
MSG_NUM	Count of current message in tuple
SIZE	Length of the ring buffer (constant)
STA_ADD	The start address of ring buffer (constant)
END_ADD	The end address of ring buffer (constant)

Let P_ε and P_τ be the priorities of etasks and threads, then by adopting the *etask-to-thread conversion* scheme, the initialization priorities of the threads of ε_H are

$$P'_{\text{cur\_}\tau}(0) = P_{\text{cur\_}\tau}(0) + (P_{\text{cur\_}\varepsilon H}(0) - P_{\text{cur\_}\varepsilon L}(t)), \quad (3)$$

where $P'_{\text{cur\_}\tau}(0)$ and $P_{\text{cur\_}\tau}(0)$ are the initial priorities of after and before conversion of τ, and $P_{\text{cur\_}\varepsilon H}(0)$ and $P_{\text{cur\_}\varepsilon L}(t)$ are the initial priority of ε_H and the current priority of ε_L.

3.3. System Communication. To simplify the system implementation, HEROS provides a uniform interface and a tuple-based communication mechanism for the interactions of system components. Basing upon the concept of parallel language *LINDA*, HEROS adopts the *tuple* space and the *IN/OUT* primitives for the message exchange and interprocess communication (IPC).

3.3.1. Tuple Space. The tuple space consists of a set of tuples (buffers), which are used to exchange data or manage signals between components. In HEROS, each thread is allocated a unique tuple, through which other components can send data to activate this thread. Two kinds of interactions are allowed for the system communication and synchronization: the interior interaction between threads and the exterior interaction between threads and peripherals.

Each tuple is allocated a critical resource that is a ring buffer, in which data are loaded at the head and read from the tail. Tuples are stored into a data table named *tuple_table*, shown in Table 3.

3.3.2. System Primitive. IN/OUT is the pair of system primitives that is responsible for the communication and data exchange between system components and peripherals. The IN primitive is called when a thread needs to read data from its related tuple. Defining μ is the tuple and τs is the source thread, the functional description of the IN primitive is shown in Figure 3.

(1) If (Data is ready in μ), then Read data from μ of τ_s, and Update the state of μ;

(2) Else, update the state of (μ, τ_s), and then call *thread_scheduler* to suspend τ_s and start a new scheduling.

The *OUT* primitive is called when an ISR (interrupt service routine) or a thread needs to communicate with one another. Defining μ is the tuple, τ_o is the object thread and

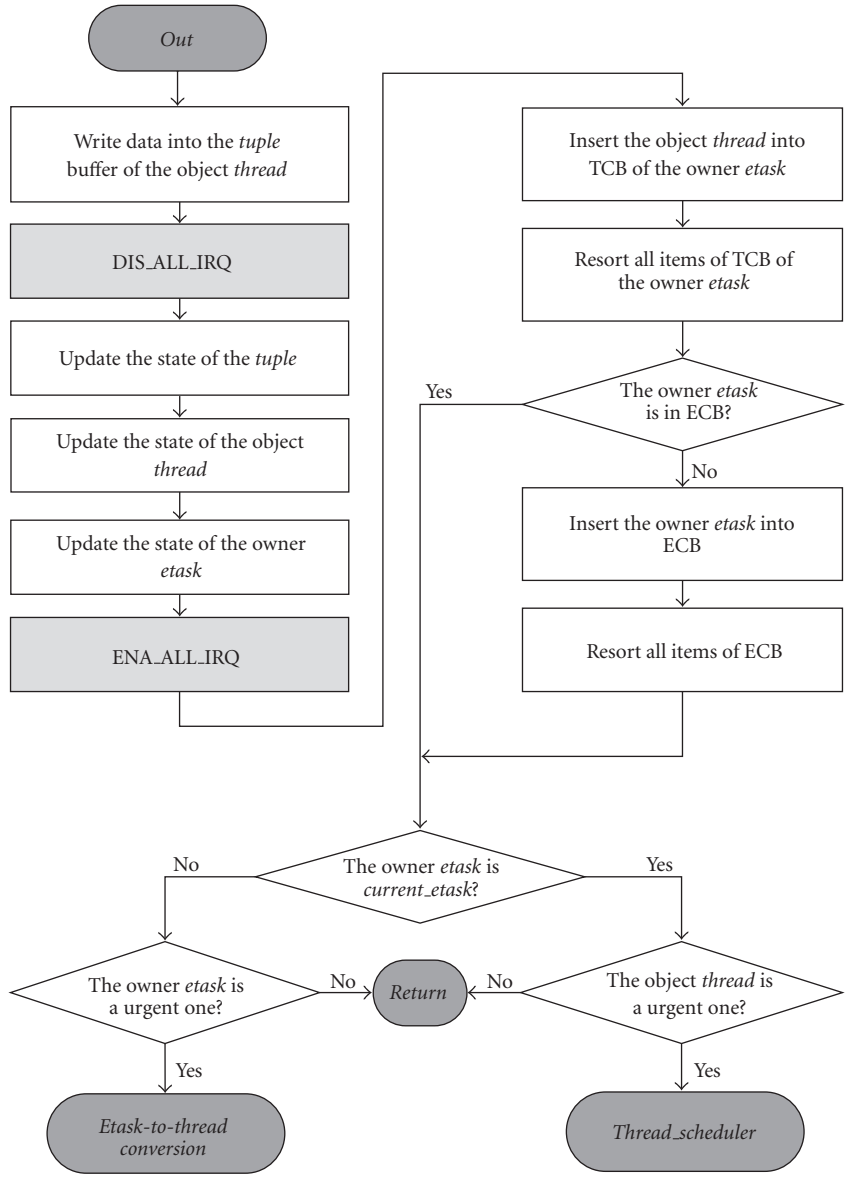


Figure 4: Functional description of the OUT system primitive.

ε_o is the owner etask, the functional description of the *OUT* primitive is shown in Figure 4.

(1) Write data into μ of τ_o;

(2) Update the states of $(\mu, \tau_o, \varepsilon_o)$;

(3) Call *InsertThreadList* to insert τ_o into TCB of ε_o, and then resort the threads of this TCB;

(4) If (ε_o is not in ECB), then call *InsertEtaskList* to insert ε_o into ECB and then resort the etasks of this ECB;

(5) If (ε_o is *current_etask* and τ_o is the highest one in TCB), then call *thread_scheduler* to start a new scheduling;

(6) If (ε_o is not *current_etask* and ε_o is the highest one in ECB), then call *etask-to-thread* to perform etask-to-thread conversion.

4. CIVIC Protocol

The design of CIVIC protocol is based on the scenarios of vehicular networks with dramatic changes of topologies according to location and time. In some scenarios, for example at night and on bad weather, the network density could get very low. In such scenarios, a communication system purely in client/server mode or in mobile ad hoc mode may not be appropriate. Since the distribution of vehicular network is generally along roads. The CIVIC assumes the roadside infrastructure MMRS (multisupport, multiservice routers and servers) can be deployed to support network access and QoS. Figure 5 shows how a message is forwarded from one node to another through mixed networking of ad hoc and infrastructure.

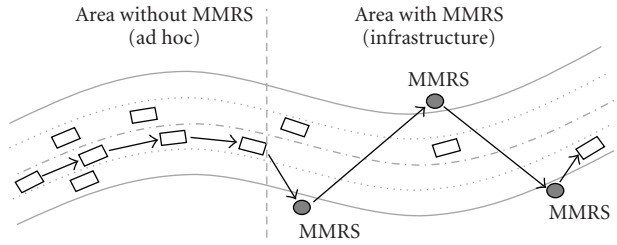


Figure 5: Mixed ad hoc and infrastructure networks.

The second assumption of CIVIC protocol is that the location and direction of network nodes could be obtained by GPS (global positioning system) on vehicles or from roadside MMRS.

4.1. Routing Mechanisms. Based on these two assumptions, CIVIC protocol is run with the following two mechanisms.

4.1.1. One-Hop Link Stability. A common way to ensure quick routing response is to keep stable connections. In a high-mobility scenario like vehicular network, the survival time of stable connections has great impact to QoS.

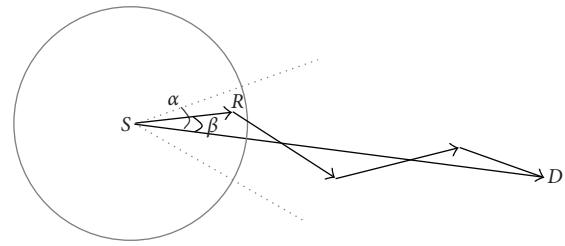
The stability of connection in CIVIC protocol is maintained by the neighbour knowledge exploration. The exploration is proactive, it is implemented by the exchange of "Hello" messages, and it must be performed only when the link stability is out of date. The dynamic interval of neighbour knowledge exploration is evaluated by $\Delta t = \text{Min}\{\Delta t_r\}$ with equation set (4).

$$\begin{aligned}
\Delta t_r &= \infty, \quad \text{if } v_r^{\max} = v_s;\\
\Delta t_r &= \frac{R + x_s - x_r^{\max}}{v_r^{\max} - v_s}, \quad \text{if } x_r^{\max} > x_s,\ v_r^{\max} > v_s;\\
\Delta t_r &= \frac{R + x_r^{\max} - x_s}{v_s - v_r^{\max}}, \quad \text{if } x_r^{\max} < x_s,\ v_r^{\max} < v_s;\\
\Delta t_r &= \frac{x_r^{\max} - x_s}{v_s - v_r^{\max}}, \quad \text{otherwise,}
\end{aligned} \tag{4}$$

where R is the radio range in the worst case; x_s is the location of source node, and v_s is its average speed; $x_r^{\max}$ is the location of one of its neighbour nodes, and $v_r^{\max}$ is the speed of this neighbour node. Both $x_r^{\max}$ and $v_r^{\max}$ are adjusted by the worst case of GPS error. The equation set (1) means that the interval of sending "Hello" messages depends on the distances and the relative speeds between the source node and its neighbour nodes.

After neighbour knowledge explorations, each node stores its neighbour information for the further multihop routing algorithm.

4.1.2. Multihop DANKAB. Due to resource constraints of embedded system and negative effects from radio irregularity [28], broadcast is a suitable transmitting scheme for IVC routing algorithm. However, it is well known that broadcast could cause serious redundancy, contention, and collision [29]. Therefore, it is important to determine a correct broadcasting technique. DANKAB (directional area neighbour knowledge adaptive broadcast) is therefore proposed.



S : Source node
R : Neighbour node of S
D : Destination node

Figure 6: DANKAB routing concept.

When the destination node is not in one-hop distance, DANKAB is used in the routing requests to find the next hop of source node. Figure 6 illustrates this process with source node S, destination node D, and routing node R. We define the direction area as an angle α with a default value of $\pm 30^\circ$. In order to reduce the number of messages in the network, only the nodes within the direction area can broadcast the message. If there is no node within the direction area, the angle α will be gradually increased (e.g., 45°, 90°, and 180°) until the next hop is found. A node can be a candidate in the next hop if $\cos\alpha \le \cos\beta$. The $\cos\beta$ is calculated by law of cosines

$$\cos\beta = \frac{\text{Dis}_{\text{sd}}^2 + \text{Dis}_{\text{sr}}^2 - \text{Dis}_{\text{rd}}^2}{2\text{Dis}_{\text{sd}}\text{Dis}_{\text{sr}}}. \tag{5}$$

In (5), Dis_{sd}, Dis_{sr}, and Dis_{rd} are the Euclidian distances between nodes S and D, S and R, and R and D, respectively. The Euclidean direction is not appropriate for defining the direction of mobile node when roads are too winding, but it can be applied for a short segment of a road.

In an infrastructure network, the roadside MMRS can provide the location of destination node D. In an ad hoc network, a location request will be performed by simple flooding to all directions. Other nodes in the same network can store the location responded from destination node to avoid resending such requests. The location of destination node may change during this process, but the DANKAB is based on broadcast, so there is no need for a very accurate location of destination node.

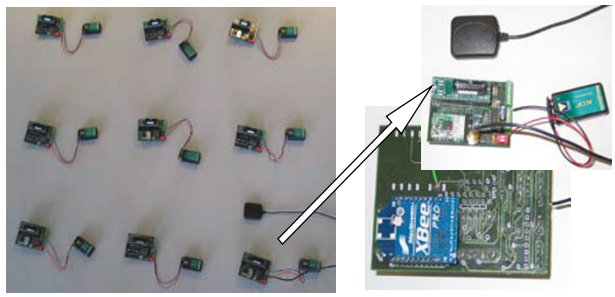


Figure 7: LiveNode platform.

When there is more than one node in the direction area, two energy-aware methods can be adopted for selecting the next candidate node. The first method is competitive broadcast. When a node in area α forwards (rebroadcasts) a routing message, it sends with a delay based on the remaining energy, thus the node with more energy will forward a message more quickly. Other nodes with less energy will discard the same routing message when they receive the first forward one. The second method is to let the source node S selecting the node for next hop. It requires the additional information about remaining energy in neighbour knowledge explorations, but it generates much less routing data. We use the second approach for the implementation in this paper.

After defining the next hop of source node S, the processes of DANKAB repeat hop-by-hop until the routing message attains the destination node or reaches the preset limitation of hop number.

If the routing path has been obtained, the data from application layer will be transmitted. If the data rate is low, DANKAB can also be integrated to the data sending, and the routing request can be ignored. For the implementation in this paper, the two mechanisms are separated.

4.2. Message Delivery Mechanisms. Based on the previous mechanisms, the CIVIC has three groups of messages as shown in Table 4.

The first group is for one-hop neighbour knowledge exploration, which includes HELLO_REQ (hello request) and HELLO_RPY (hello reply) messages. The second group is for multihop routing request and reply preformed by DANKAB. The ROUTE_REQ_SF is sent when the location of destination node is unknown. This message is normally reply by ROUTE_RPY_CIVIC. The ROUTE_REQ_CIVIC message is sent when the location of destination node is known, and it is normally replied by ROUTE_RPY_BY_PATH. More details of routing message will be described in the next part.

The data from application layer is contained by a DATA_SEND_BY_PATH message. To assure such message reaches the destination node, a node can ask the destination node to send back a acknowledge message, which is named DATA_ACK_BY_PATH.

5. System Design

5.1. Hardware. Our hardware platform is LiveNode [11], a versatile wireless sensor platform that enables to implement rapidly a prototype for different application domains. The LiveNode hardware platform has been successfully used in applications including telemedicine (wireless cardiac arrhythmias detection), intervehicle communication [30], and environmental data collection (FP6 EU project NeT-ADDED).

Table 4: Message groups.

Group	Name	Max Size (Byte)
Hello	HELLO_REQ	29
	HELLO_RPY	25
Routing	ROUTE_REQ_SF	28
	ROUTE_REQ_CIVIC	29
	ROUTE_RPY_CIVIC	48
	ROUTE_RPY_BY_PATH	28
Application	DATA_SEND_BY_PATH	64
	DATA_ACK_BY_PATH	12

As shown in Figure 7, the LiveNodes used for the experiments have the three major modules including an Atmel AT91SAM7S256 microcontroller (ARM7TDMI core), a MaxStream XBee Pro chip to ensure wireless communications on 802.15.4 standard, and a GlobalSat ET-301 GPS chip for specific GPS signal/data processing.

5.2. Software. The new embedded system can provide adaptive task mechanisms for different IVC application requirements. This section describes an event-driven software design that has been tested. In this design, the flow of computing process is driven by OS events such as packet arriving, location updating, and timer noticing, thus it can complement protocol stack works and leave low memory footprint [31].

Figure 8 demonstrates the system stack and the event-driven data flow. There are four major event-driven etasks in the system. The TIMER_RDY etask is driven by interrupts from the microcontroller PIT (periodic interval timer). The rest of etasks are mainly driven by interrupts from the USART (universal synchronous/asynchronous receiver/transmitter) ports connected to GPS module (US0) or XBee module (US1). Figure 9 shows an example of processing flow between etasks and threads. Only the etasks and threads relating to the major system process are shown in Figures 8 and 9.

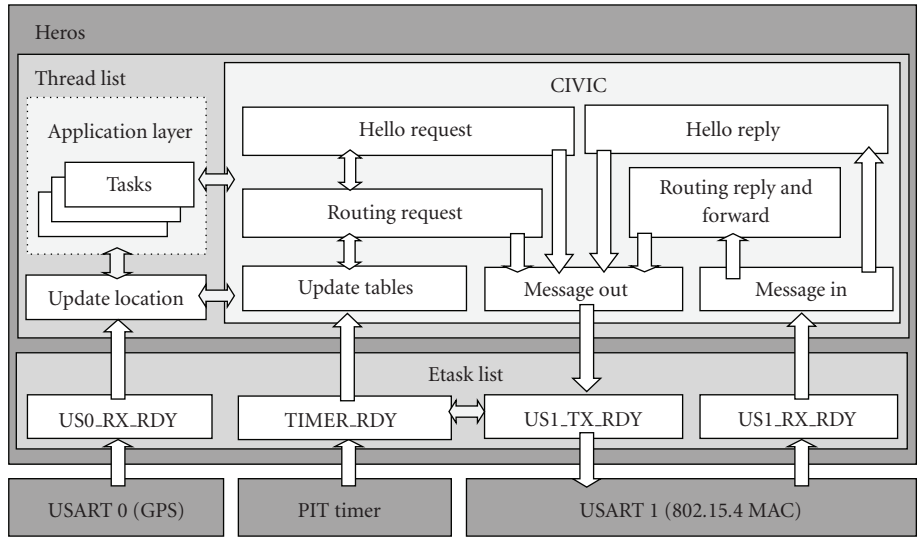


Figure 8: The system stack and the event-driven data flow.

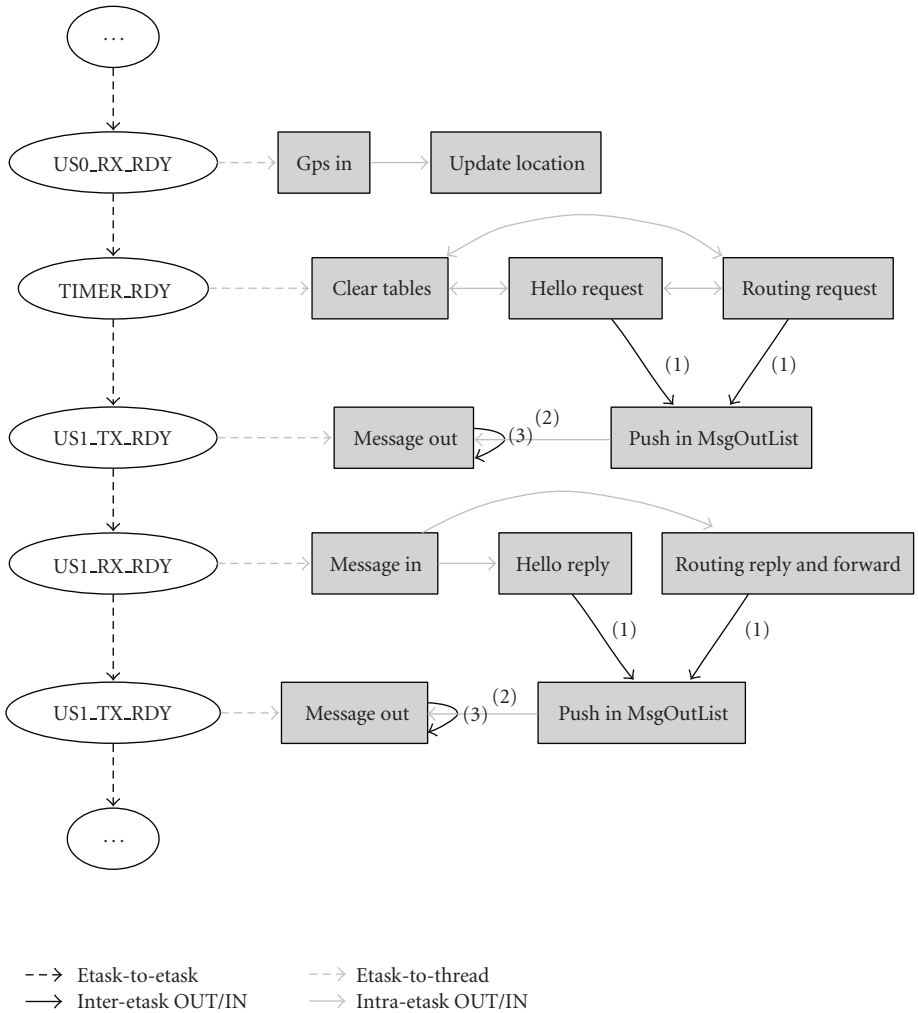


Figure 9: The interactions between etasks and threads.

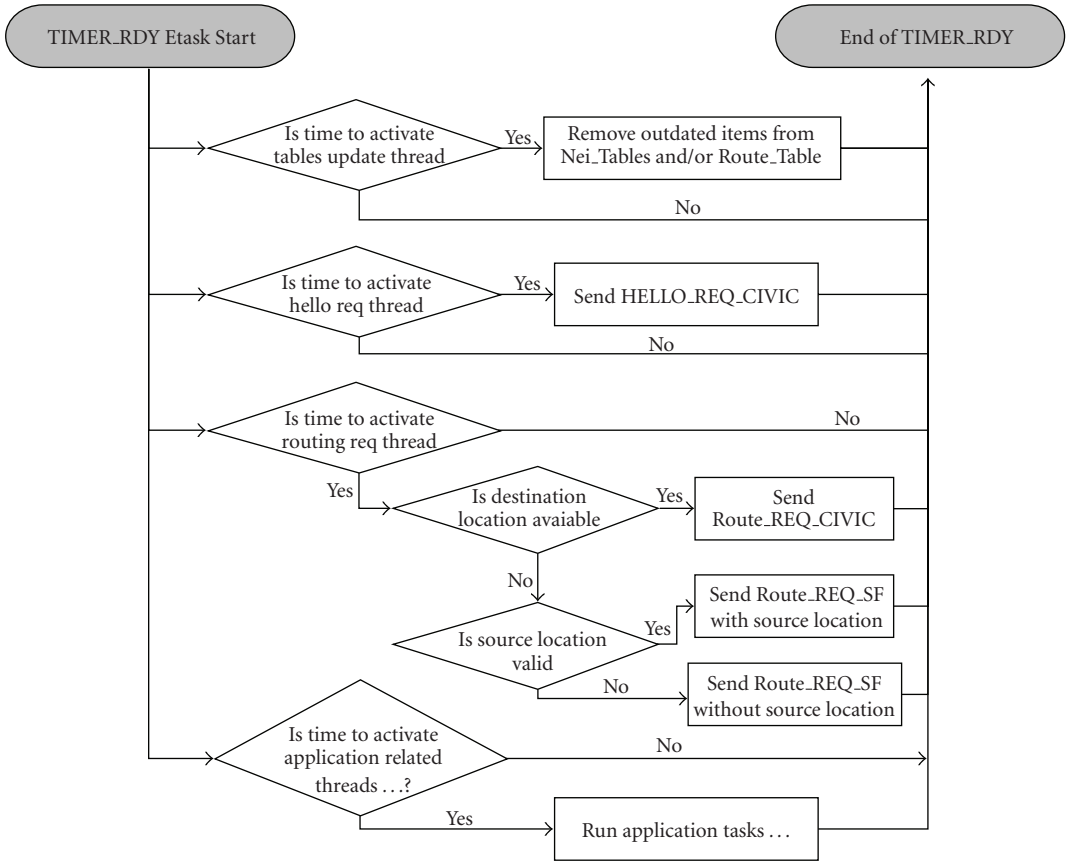


Figure 10: Dataflow of TIMER_RDY Etask.

The TIMER_RDY etask runs the periodic tasks, for example, sending "Hello" messages, activating proactive routing searches, and removing the outdated table items. The tables need to be cleared periodically are the neighbour table and the routing table. Figure 10 gives a zoom-in vision of the TIMER_RDY etask.

The US1_TX_RDY etask handles the message outputs. To avoid the sending intervals becoming too short, other etasks should not directly send out messages. Instead, they push messages into a buffer list called MsgOutList as the step one shown in Figure 9. It will activate the US1_TX_RDY etask to check whether the last transmission has been finished. If it has been finished, a message will be sent out by the "Message Out" thread (Step 2); if not, the etask is end, and a PIT timer will be activated to run the etask after a waiting period (Step 3). In addition, for the time-sensitive designs, the TIMER_RDY etask can take control of the output related to send message at a fix interval.

The etasks US0_RX_RDY and US1_RX_RDY contain threads to process incoming raw data. The former deals with the GPS data, the latter deals with the CIVIC data. The major routing for these two etask is similar: (1) when the input buffer is ready for data processing, a thread translates the raw data into meaningful messages; (2) based on the message types, the etask divide messages into the related threads for further actions. Figure 11 shows the actions in a US1_RX_RDY etask. In addition, Figures 10 and 11 demonstrate the message delivery mechanism of CIVIC protocol described in the last section.

6. System Evaluation

6.1. HEROS Evaluation

6.1.1. Memory Consumption. HEROS is dedicated to strict resource-constrained mobile devices and embedded applications, it should have small resource consumption, especially the memory consumption. Table 5 shows the memory consumption of main functions in HEROS.

6.1.2. Execution Time of IN/OUT Primitives. The deterministic and predictable behaviours of system primitives are the key features of a real-time operating system. In HEROS, the execution time of system primitives is determined and bounded between the minimal and maximal values. Table 6 presents the performance evaluation of *IN/OUT* primitives at 48 MHz.

In the *IN* primitive, the maximal value is the execution time of reading n bytes from the thread tuple when a message is ready; the minimal value is the execution time of calling *thread_scheduler* when no data is available. In the *OUT* primitive, the execution time is the time interval of writing n bytes into the thread tuple and then calling *InsertThreadList*. The message length n is limited between 0 and the length of

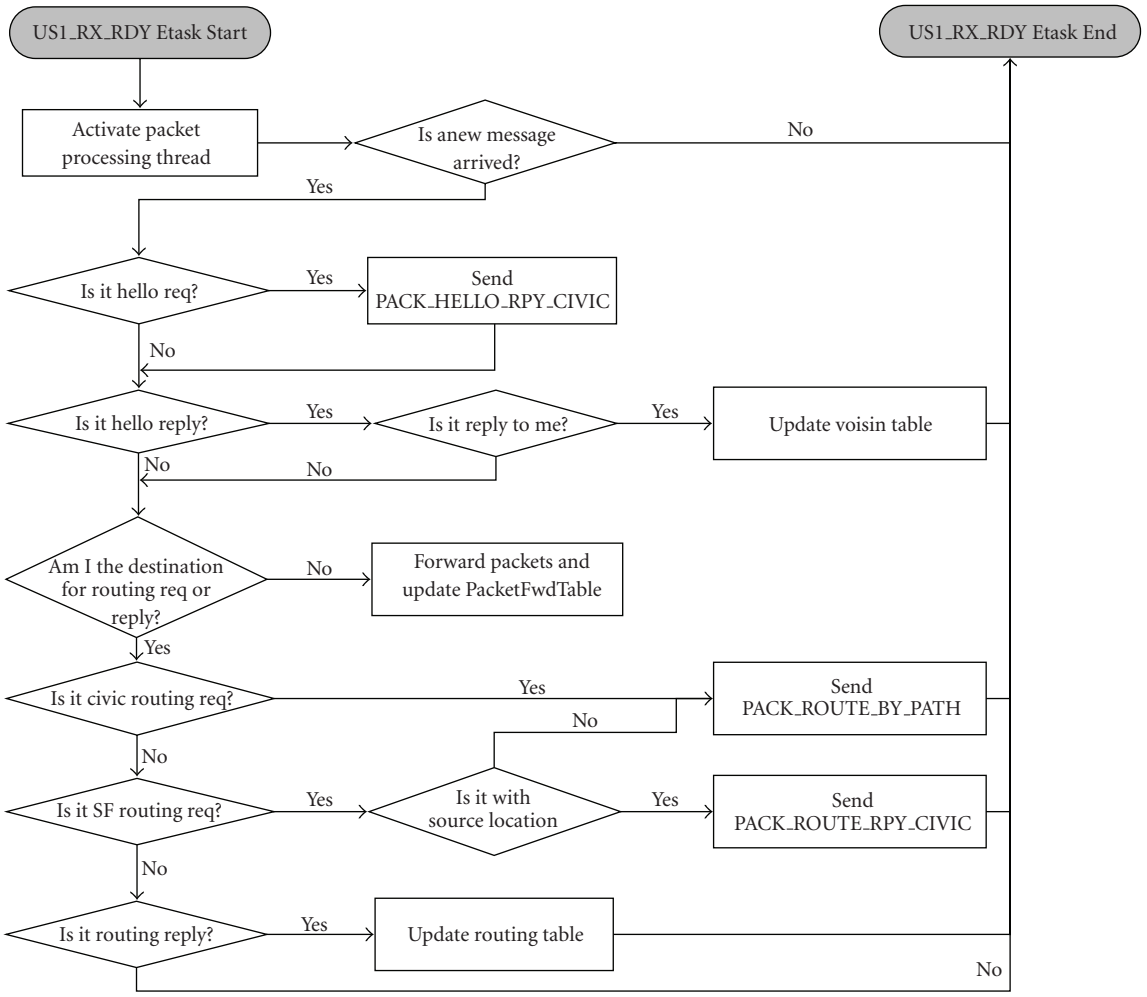


Figure 11: Dataflow of US1_RX_RDY Etask.

Table 5: Memory consumption of HEROS.

System functions		*Memory consumption (BYTE)*
In		160
Out		124
Etask_Manager		152
Thread_Scheduler		216
InsertThreadList		152
DeleteThreadList		158
InsertEtaskList		184
DeleteEtaskList		152
EtasktoThread		92
Total	CODE	3572
	DATA	1272

the tuple buffer ($0 < n < N$), thus the execution time of IN/OUT primitives are also predictable.

6.1.3. System Latencies. The mechanisms of system scheduling and interrupt handling are critical for the designs of RTOSs, which should promise the predictable and deterministic system behaviours. System latencies are the key metrics to evaluate the real-time performance of RTOSs. In HEROS, the *etask-to-etask switch latency* and *thread-to-thread switch latency* are used to evaluate the system scheduling mechanism, and the *interrupt response latency* and *interrupt dispatch latency* are used to rate the interrupt handling mechanism.

(i) *etask-to-etask switch latency*: the time elapsed for system switching from one etask to another. The *etask_manager* is called to run the next ready *etask*. The belonging thread of this *etask* is activated in the "cold" mode.

Table 6: Execution time of IN/OUT primitives.

Primitive		In	Out
Cost (cycles)	Max	$149 + 46\,n^*$	$104 + 32\,n$
	Min	95	
Time (μs) (48 MHz)	Max	$3.101 + 0.957\,n$	$2.164 + 0.666\,n$
	Min	1.977	

*: n is the data length (byte), $0 < n < N$, (N = the buffer length − 2);

Table 7: Performance evaluation of system latencies.

Latencies		Cost (cycles)	Time (μs) (48 MHz)
Etask-to-etask switch		90	1.873
		Call *etask_manager* to run the next *etask*. The ready *thread* of this *etask* is performed in "cold" mode;	
Thread-to-thread switch		89[1] (99[2])	1.852[1] (2.06[2])
		Call *thread_scheduler* to run the next *thread* [1]: *thread* is performed in "cold" mode; [2]: *thread* is performed in "warm" mode.	
Interrupt response	Min.	29	0.604
	Avg.	107	2.227
	Max.	247	5.140
Interrupt dispatch	Min.	32	0.666
	Avg.	68	1.415
	Max.	176	3.662

(ii) *thread-to-thread switch latency*: the time elapsed for system switching from the current thread to another. The *thread_scheduler* is called to run the next ready *thread*. Note that the *thread* can be activated in the "code" or "warm" mode.

(iii) *Interrupt response latency*: the time elapsed from the last instruction of the interrupted components and the first instruction in ISRs, which indicates the rapidity of the system reaction to an external interrupts.

(iv) *Interrupt dispatch latency*: the time interval to go from the last instruction in the interrupt service routine to the next thread scheduled to run, which indicates the time needed to switch from interrupt mode to user mode.

The evaluation results of system switch latencies and interrupt latencies are presented in Table 7. The performance results show that HEROS has deterministic system switch/interrupt latencies satisfying the requirements of most of real-time applications. Note that the latencies of system switch are fixed with no concern of the numbers of *etask* (thread).

6.1.4. Comparison with TinyOS. TinyOS has been tested on ATmega128 (4 MHz) [32] and HEROS is tested on AT91SAM7S256 (48 MHz). This paper only compares three system operations (i.e., scheduling a task, context switch and hardware interrupt latency) and the memory consumption between HEROS and TinyOS, shown in Table 8. Note that the operation of context switching happens between the two threads in the "warm" mode. In order to support real-time multitask operations, HEROS has more system overheads than TinyOS but has similar system cycles for the basic system operations.

6.2. CIVIC Evaluation. The first experiment is done in a network with nine static sensors. A sensor is set as the destination node. The other sensors keep sending routing requests and application data to this destination sensor. We implement all mechanisms of CIVIC protocol on HEROS for the experiment. The maximum interval of US1_TX_RDY for message sending is set to be in 100 ms.

Table 8: Performance evaluation between HEROS and TinyOS.

Operations	HEROS (AT91SAM7S256)		TinyOS (ATmega128)	
	Cost (cycle)	Time (μs)	Cost (cycle)	Time (μs)
Scheduling a task	43	0.895	46	11.5
Context Switch	56	1.165	51	12.75
Hardware Interrupt (hw)	5	0.104	9	2.25
Hardware Interrupt (sw)	61	1.269	71	17.75
OS Size (byte)				
CODE	3572		1272	
DATA	432		48	

Since the sensors are static, we replace the GPS location with a random location in a parking (about 80×60 meter squares) of our campus, and sensors only response to messages sent from the distance less than 50 meters. Because each sensor actually receives all messages from the network, it requires more operations by both CIVIC and HEROS. This experiment focuses on testing the overall network performance.

The experiment contains four identical subtests, each runs 15 minutes, and then gets average values. Table 9 indicates the network performance. The loss rate is calculated basing upon the missing serial number. When a message outputs to the network by a sensor, the CIVIC protocol attaches a serial number (one byte) to indicate its place in the message sequence of the sensor. By monitoring the serial number, we know the message loss rate for every sensor.

Table 9: Individual sensor status.

	Average	S0	S1	S2	S3
Message number	**2973**	2207	3074	2995	3203
Message rate	**3.41**	2.46	3.63	3.67	3.57
Loss rate	**4.13**%	0.85%	6.54%	9.84%	1.02%
	S4	S5	S6	S7	S8
Message number	3130	2734	3536	3190	2689
Message rate	3.51	3.14	3.90	3.64	3.20
Loss rate	1.51%	3.94%	2.77%	3.33%	7.34%

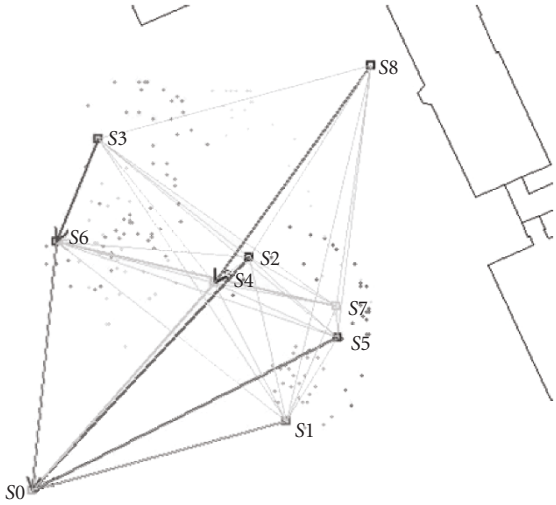


Figure 12: Data sending flows to a destination node.

In general, most of sensors are sending routing requests and application data to the accurate direction. Figure 12 shows an example of CIVIC routing topology. The S0 at the corner is the destination node.

The message loss rate in this experiment should be caused by network traffic overhead, interferences, shading, and message collisions. As previous mention, all nine nodes are actually receiving message from the whole network, so they have the same conditions of the incoming network traffic. Moreover, we can assume that the message rate in Table 9 indicates the intranode performance of single node. If a node cannot handle such incoming network traffic and cause output message loss, the loss rate should be increased with the message rate.

The implementation of first experiment consumes 42,736 K bytes memory without any optimization from compiler as shown in Table 10. DATA can be less than the shown one because the experiment needs to allocate extra memory to keep results. The memory consumption is majorly from four sources: CIVIC protocol, GPS data processing, HEROS microkernel, and hardware drivers.

The next experiment is to test the message delay and bandwidth capability by excluding the network related efforts. The experiment is done with two static sensors. One sensor acts as a message sender. It keeps sending

Table 10: Memory consumption of this system.

	Size (byte)
CODE	26,356
DATA	14,746
CONST	1,634
Sum	42,736

new messages with growing size and interval. The other sensor acts as a receiver, and it acknowledges the incoming message. The size of acknowledge is 5 bytes. If the sender does not receive the acknowledgment of last message, the same message will be resent three times. The size of message increases from 5 bytes to 245 bytes. The interval of sending message increases from 10 ms to 100 ms. All subtests perform 10 times and get average values. This experiment only enables etasks for sending and receiving. The message delay and resending count are used to evaluate network performances. The "delay" in this paper means the timing between sending a new message and receiving its acknowledgment.

The results of this experiment are shown in Figures 13 and 14. The experiment shows the limitation of this embedded system.

Firstly, the results on message delay may not reflect on the results of loss rate. It only indicates the US1_RX_RDY etask cannot get access to its processing thread because the US1_TX_RDY etask is too busy on sending message. The design of etasks does not allow preemptive, thus it is normal.

Secondly, the increase rate of message delay is slow down after the message size reaches about 100 bytes. The reason is from the transmitting mechanism of Xbee clip and the sequence of delay factors. The Xbee clip has an internal 100 bytes output buffer. When the message size is lower than 100 bytes, the packet will not be sent immediately. Thus, the delay factors in a transmitting happen separately and in succession. There are three major delay factors from the transmissions in sender, over-the-air, and in receiver. When the size of messages is more than 100 bytes, these three factors happen simultaneously and the overall delay decreases. Since the maximum message size of CIVIC protocol is less than 64 bytes (at 100 ms sending intervals), the result between 5 to 65 bytes is more important for evaluating the system.

Thirdly, we can know from Figure 13 there is an initial delay for all message sending. Figure 15 zooms in on the initial delay. In the first column, we can find the delay in sending is already 5 ms. However, each time when the message size increases 5 bytes, the delay in sending only increases 1-2 ms. Since only 1-2 ms is used in actual data transmitting, the initial delay before transmitting is about 3-4 ms. The initial delay is from two sources: (1) after data arrives to the buffers inside a XBee chip, the XBee chip need to have its data processing; (2) the CSMA-CA in 802.15.4 standard needs time to sensing channels before transmitting data. In addition, the bandwidth is about 5 bytes/ms between 5 bytes to 65 bytes. It is the information rate, not the raw data rate.

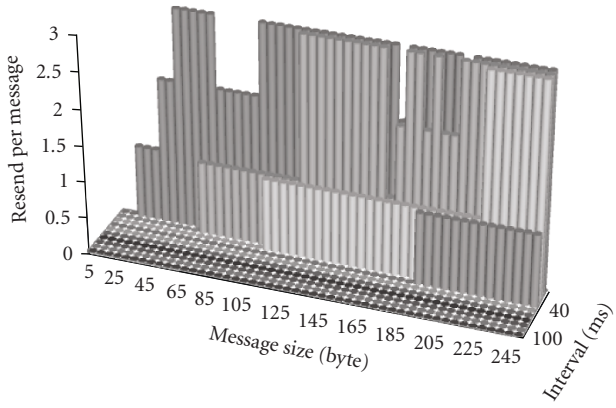


Figure 13: Resend counts of all subtests.

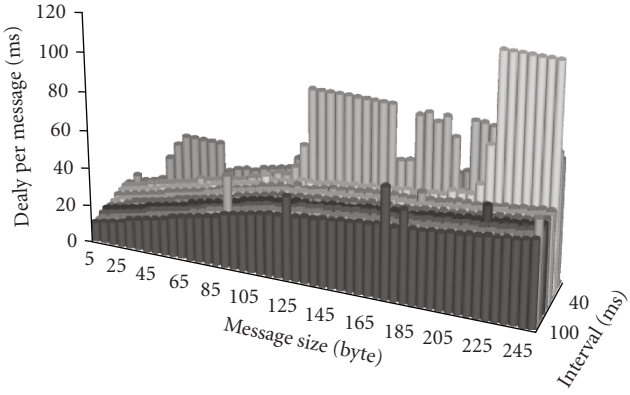


Figure 14: Message delays for all subtests.

7. Conclusion and Ongoing Work

Embedded communication systems normally contain two basic component layers: a protocol stacks to manage network communications and an operating system to interface with hardware and schedule tasks or events. Based on such structure, this paper presents a new low-cost and low-memory footprint design and its implementation for embedded IVC applications with CIVIC as protocol stack, and HEROS as embedded OS.

HEROS proposes an Etask/Thread modular architecture and adopts a tuple-based IN/OUT primitive communication mechanism to provide both event-driven and real-time multitasking operation modes. CIVIC adopts the DANKAB mechanisms to provide a resource-awareness and rapid convergence routing algorithm. By these designs, this system can adapt to a wider range of IVC applications. At present, the CIVIC protocol has been ported on HEROS to perform the real-time multitasking and event-driven WSN applications. The experiment results in Section 5 show that this embedded system has small resource consumption and is adaptable to different applications. Moreover, thanks to low message-sending delay, the present design may be used to implement low cost embedded collision avoidance device by combining GPS receiver and IVC data.

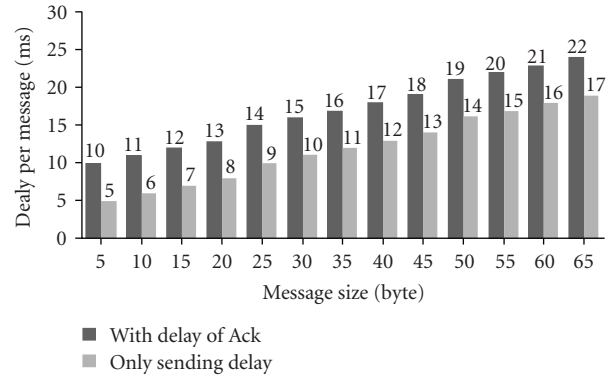


Figure 15: Delay per message between 5 and 65 bytes.

Currently, this low-cost embedded system is still under development and evaluation, the result of network performances is not so perfect, but it shows the adaptability of this system to high-mobility scenarios.

The ongoing works of CIVIC are to reduce memory consumption and continue improving its communication QoS. For HEROS, its programming models are not completed, thus a hardware abstract layer or interface will be provided in future. Moreover, the current implementation will be carried out on the intelligent multimodal transportation system in Clermont-Ferrand city (France).

Acknowledgments

The authors would like to thank all colleagues who contributed to the study. The authors would also like to thank the reviewers for their remarks, which enable to improve this paper. The authors are grateful for the EU (NeT-ADDED FP6 EU project) and the French EGIDE international cooperation plan (PHC PFCC No. 20974WG) for their supports to this project.

References

[1] B. Fitzgibbons, R. Fujimoto, R. Guensler, M. Hunter, A. Park, and H. Wu, "Simulation-based operations planning for regional transportation systems," in *Proceedings of the 5th Annual National Conference on Digital Government Research: New Challenges and Opportunities*, Seattle, Wash, USA, May 2004.

[2] Y. Shiraki, et al., "Development of an inter-vehicle communications system," *OKI Technical Review 187*, vol. 68, pp. 11–13, 2001.

[3] M. Käsemann, et al., "A Simulation Study of a Location Service for Position-Based Routing in Mobile Ad Hoc Networks," 2002, http://www.et2.tu-harburg.de.

[4] M. Wurpts and R. Logan, "HLA inside and out: intra- and inter-vehicle communications," in *Proceedings of the Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC '03)*, p. 9, 2003.

[5] X. Diao, M. Kara, J. Li et al., "Experiments on PAVIN platform for cooperative inter-vehicle communication protocol (CIVIC)," in *Proceedings of the IEEE Africon Conference*, Nairobi, Kenya, September 2009.

[6] X. X. Diao, et al., "Cooperative inter-vehicle communication protocol with low cost differential GPS," *Journal of Networks*, vol. 4, no. 6, pp. 445–457, 2009.

[7] T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol," IETF RFC 3626, October 2003.

[8] T. Kosch, Ch. Schwingenschlögl, and L. Ai, "Information dissemination in multihop inter-vehicle networks—adapting the ad-hoc on-demand distance vector routing protocol (AODV)," in *Proceedings of the 5th IEEE International Conference on Intelligent Transportation Systems*, Singapore, 2002.

[9] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends," *Computer*, vol. 19, no. 8, pp. 26–34, 1986.

[10] D. Gelernter, "Generative communication in LINDA," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[11] K. M. Hou, G. De Sousa, J. P. Chanet, et al., "LiveNode: LIMOS versatile embedded wireless sensor node," in *Proceedings of the 7th International Conference on New Technologies of Distributed Systems (NOTERE '07)*, pp. 65–69, Marrakech, Morocco, June 2007.

[12] K. Raatikainen, "Operating system issues in wireless ad-hoc networks," in *Proceedings of the Keynote Speech in International Workshop on Wireless Ad-Hoc Networks*, May 2005.

[13] H. Y. Zhou, K. M. Hou, and C. De Vaulx, "SDREAM: a super-small distributed REAl-time Microkernel dedicated to wireless sensors," *International Journal of Pervasive Computing and Communications*, vol. 12, no. 4, pp. 398–410, 2006.

[14] J. J. Labrosse, *MicroC/OS-II: The Real-Time Kernel*, CMP Books, 2nd edition, 2002.

[15] P. Levis, S. Madden, J. Polastre, et al., "TinyOS: an operating system for sensor networks," in *Ambient Intelligence: Part II*, pp. 115–148, Springer, Berlin, Germany, 2005.

[16] R. Barr, J. C. Bicket, D. S. Dantas et al., "On the need for system-level support for ad hoc and sensor networks," *ACM Operating System Review*, vol. 36, no. 2, pp. 1–5, 2002.

[17] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki—a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN '04)*, pp. 455–462, November 2004.

[18] S. Bhatti, J. Carlson, H. Dai et al., "MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 563–579, 2005.

[19] S. Dulman and P. Havinga, "Operating system fundamentals for the EYES distributed sensor network," Progress Report, Utrecht, The Netherlands, 2002.

[20] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys '05)*, pp. 163–176, 2005.

[21] C. E. Perkins and P. Bhagwat, "Highly Dynamic Destination Sequence-Vector Routing (DSDV) for mobile computers," *Computer Communication Review*, vol. 24, no. 4, pp. 234–244, 1994.

[22] R. Ogier, F. Templi, and M. Lewis, "Topology dissemination based on reverse-path forwarding (TBRPF)," IETF RFC 3684, February 2004.

[23] D. B. Johnson, D. A. Maltz, and J. Broch, "DSR: the dynamic source routing protocol for multi-hop wireless ad hoc networks," in *Ad-Hoc Networking*, C. E. Perkins, Ed., pp. 139–172, Addison-Wesley, New York, NY, USA, 2001.

[24] V. D. Park and M. S. Corson, "Highly adaptive distributed routing algorithm for mobile wireless networks," in *Proceedings of the 16th IEEE Annual Conference on Computer Communications (INFOCOM '97)*, pp. 1405–1413, April 1997.

[25] C. K. Toh, "Long-lived ad-hoc routing based on the concept of associativity," IETF MANET Working Group, Internet Draft, March 1999.

[26] R. Dube, C. D. Rais, K.-Y. Wang, and S. K. Tripathi, "Signal stability-based adaptive routing (SSA) for ad hoc mobile networks," *IEEE Personal Communications*, vol. 4, no. 1, pp. 36–45, 1997.

[27] J. Schaumann, "Analysis of the Zone Routing Protocol," December 2002, http://www.netmeister.org/misc/zrp/zrp.pdf.

[28] G. Zhou, T. He, S. Krishnamurthy, and J. A. Stankovic, "Impact of radio irregularity on wireless sensor networks," *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services (MobiSys '04)*, pp. 125–138, 2004.

[29] Y.-C. Tseng, S.-Y. Ni, and E.-Y. Shih, "Adaptive approaches to relieving broadcast storms in a wireless multihop mobile ad hoc network," *IEEE Transactions on Computers*, vol. 52, no. 5, pp. 545–557, 2003.

[30] H. Y. Zhou, G. De Sousa, J.-P. Chanet et al., "An intelligent wireless bus-station system dedicated to disabled, wheelchair and blind passengers," in *Proceedings of the IET International Conference on Wireless Mobile and Multimedia Networks (ICWMMN '06)*, p. 434, November 2006.

[31] M. Moubarak and M. K. Watfa, "Embedded operating systems in wireless sensor networks," in *Guide to Wireless Sensor Networks*, S. Misra, I. Woungang, and S. C. Misra, Eds., pp. 323–346, Springer, New York, NY, USA, 2009.

[32] W. Maurer, "The Scientist and Engineer's Guide to TinyOS Programming," 2004, http://ttdp.org/tpg/html.