Transplantable CANopen Master Based on Non-preemptive Task Scheduler*

Wei Song, Shizhen Yan, Zhe Xu, and Suiming Fang

College of Electronic Information and Control Engineering
Beijing University of Technology
100 Ping Le Yuan, Beijing 100022, P.R.China

{benjaminweber, shizhenyan}@emails.bjut.edu.cn, {xu2002, suiming}@bjut.edu.cn

***Abstract* – Since CANopen master is required to run real-timely, concurrently and dynamically, multi-task solution is a feasible choice. An ANSI C non-preemptive task scheduler is proposed, and then a transplantable and standard compatible CANopen master is implemented. Experimental results show that CANopen master based on non-preemptive scheduler meets timing constraints of internal control system of hybrid electric vehicles.**

***Index Terms* – Controller Area Network (CAN), real-time scheduling, distribute real-time system, embedded systems.**

I. INTRODUCTION

CANopen is an application layer protocol based on the Controller Area Network (CAN) serial bus system [1]. It provides several promising features. A detailed defined group of entries − object dictionary (OD) is the kernel of CANopen. Every node in CANopen network possesses an OD in which all communication parameters and processing data are stored. Service data object (SDO) defines a communication by which a node can read or write OD entries of other nodes, a way to configure network. Process data object (PDO) provides a real-time and synchronous means to exchange data among nodes. Besides, CANopen defines synchronisation object (SYNC) and timer object to synchronize all nodes, and network management (NMT) messages to control and trace node statuses.

Since CANopen is partially free of charge and easy to be applied into industrial control systems, it has been widely employed in numerous applications equipped with internal control networks. Especially, the CANopen network is expected to be used in the internal control system of the hybrid electric vehicle (HEV).

II. TASK SCHEDULER

A. Field Requirements of CANopen Master

CANopen master is the master node performing a combined role of NMT master, SYNC producer, time producer, even the SDO manager and configuration manager [2]. Required by network features, master node runs real-timely, concurrently and dynamically.

1) Real-time: According to basic timing specifications in [1], communication period is measured in microseconds. Also estimated in [3], data exchange rate requirement for various modes in HEV control system is of the order of 500 microseconds to one millisecond. Thus, 500 microseconds is the foreseeable minimal communication period.

2) Concurrency: Fulfilling the role of several managers in CANopen network, master node processes SDO, PDO and NMT messages from all slave nodes simultaneously. During network boot-up stage, master node paralleled launches boot-up check for each slave node [2]. Under operational mode, arrivals of PDOs are randomized by their transmission types, and this randomization causes message bursts and system variable jitters [4]. Although some message scheduling algorithms [4, 5] have been proposed to alleviate these problems, they intensely rely on the processing capability of concurrent messages.

3) Dynamic Adjustment: Unlike slave nodes, master node has no foresight of the number of SDO and PDO pairs required. A new entering node or a new PDO pair configured by high-level control software definitely change the PDO and SDO pairs during run-time. Therefore, undefined number of slave nodes and run-time configurations make the dynamic run-time adjustment a must.

B. Method of Task Scheduling

Common operating systems use thread scheduling to meet task response deadlines [6]. The thread naturally depicts a processor for an event. Besides, it is easy to add new threads into system. However, some schedulers provided by common operating systems do not satisfy timing requirements, one message per 500 microseconds. Systems like Linux and Windows do not support microsecond level scheduling. Besides timing issues, it is extremely complicated to transplant a multi-thread program from one operating system to another, while master node in HEV system can reside on different platforms under different conditions, e.g. it can be sited in the central monitor running WinCE, or in the vehicle's central controller without an operating system.

Hence, a task scheduler implemented by pure ANSI C is proposed. Because this scheduler only uses ANSI C, it runs on any platforms that provide C compiler. Besides, it merely realizes the CANopen related features, thus it runs faster than normal scheduling algorithms. Admittedly, a manually designed scheduler increases coding burden and it is almost impossible to build preemptive scheduling methods without

* This work is supported by combined grand from Beijing Education Committee and Beijing Science Foundation #KZ20041000501.

the help of assembling languages. Nevertheless, following part of this paper will prove that, CANopen master calls for simple scheduling algorithms and structures. Meanwhile, the non-preemptive scheduler meets timing requirements.

C. Task Object

The basic idea in task scheduler is enclosing an event procedure into a data structure, which can be inserted into and deleted from queues, like the task_struct in Linux [7]. Here, it is called *task object*

Task object comprises of necessary components to implement a minimal scheduler. Fig. 1 provides the detailed definition.

```
typedef struct _TaskObj {
   char runPrio;      /* priority */
   char * pRunEvent;  /* event trigger */
   char (*pFun) (struct _TaskObj *);   /* task function */
   struct _TaskObj * pNext;     /* doubly linked list */
   struct _TaskObj * pPre;      /* doubly linked list */
   long Argu[10];     /* data space */
   long timeHigh;     /* time stamp high part */
   long timeLow;      /* time stamp low part */
} TaskObj, *pTaskObj;
```

Fig. 1 Definition of *task object*

1) Priority: *Task object* provides eight different priorities, from the highest "0" to lowest "7", stored in runPrio. They tightly relate to the event types.

TABLE I
DEFINITION OF PRIORITIES

Priority	Type of Task
0	CAN driver and SYNC Producer
1	PDO message distributor
2	PDO processor
3	SDO message distributor
4	SDO Processor
5	NMT Error Control distributor
6	Heart beat and node guard processor
7	User interface

CANopen is a strictly priority scaled protocol, which can be derived from the definitions of function identifications [1]. As a result, fixed priorities to different message processors will keep messages of higher priority always being treated before messages of lower priority. That is, emergency messages are directly analyzed by CAN driver, PDO messages are mapped before SDO messages, and NMT status messages are checked only other messages are processed.

2) Trigger: Time and event are two kinds of triggers in this scheduler.

Obviously, CANopen especially emphasizes time request. Every message more or less has its timing constraints, and some of these constraints are accurate to microsecond. To avoid overflow, *task object* defines a 64-bit time stamp (timeHigh and timeLow) recording the microseconds past from power up. All time stamps are synchronized to a kernel timer, and the stamp of every *task object* blocked in waiting queue represents the resume time in future. Therefore, by comparing stamps in waiting queue with kernel timer, scheduler moves resumed tasks from waiting queue to runnable queue, only one accurate timer is required.

Event is the other trigger. Commercial operating systems define events by complex structures, even assembling languages. To avoid the transplanting issues brought by assembling languages, *task object* uses a pointer runEvent to represent events. When the number runEvent pointing to is non-zero, event is set, and task is triggered. Admittedly, this method cannot trigger tasks immediately after events, but pure ANSI C cannot implement a preemptive scheduler either. The non-preemptive method does not need immediate triggers.

Both time stamp or event can trigger a task. Combining them, this scheduler realizes conditional triggers like cycled run, waiting an outside event, mutual synchronization between tasks, and timeout when waiting for an event.

3) Task Function and Data Space: In every *task object*, pFun points to the function executing the task. Meanwhile, since scheduler may block tasks waiting for next trigger, task function needs its own exclusive data space to save its involatile data, an integer group Argu.

4) Doubly Linked List: Like Linux kernel [7], *task object* uses the doubly linked list structure. Since there are numerous queue operations during scheduling process, and doubly linked list costs fixed time when adding and deleting nodes, it saves operating time when task number is huge.

D. Scheduling Algorithm

There are two important queues in CANopen Master: waiting queue and runnable queue. Waiting queue is a doubly linked list, which stores all blocked *task objects*. Runnable queue stores all resumed *task objects*. Once a running task voluntarily releases or blocks itself, scheduler finds and launches a task with the highest priority in runnable queue.

Unlike waiting queue, runnable queue has eight sub-queues, illustrated in Fig. 2. Every sub-queue stores *tasks objects* with the same priority, and corresponding flag indicates queue's non-empty. Under this structure, the job of finding *task object* with the highest priority is just finding the first non-empty sub-queue from queue one to queue eight, then the first *task object* in this sub-queue is the next task to run. Demonstrated in [7, 8], as searching time is irrelevant to the number of *task objects* in runnable queue, it is an O(1) algorithm to find next task to run.

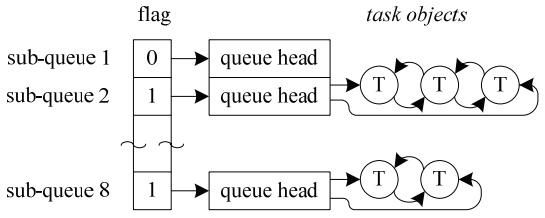

Fig. 2 Runnable queue

However, scheduler needs to scan all *task objects* in waiting queue. As mentioned above, triggers are defined by pRunEvent, timeH and timeL in *task objects*, and it is

scheduler's job to move every newly resumed task from waiting queue to corresponding sub runnable queue. Scheduler must check the triggers of every task object in waiting queue. Although several methods are proposed [7, 9-11] to mix trigger checking and task searching, they induce huge coding burden which is unnecessary in CANopen.

The whole scheduling algorithm is shown in Fig. 3. Firstly MoveTaskToRun() scans all *task objects* in waiting queue to move resumed *task objects* to runnable queue. This is an O(n) algorithm, while n represents the number of *task objects* blocked in waiting queue. Then, SelectTaskForRun() uses fixed time method finding the next runTask and executes pFun of it. When task returns, according to its return value, TASK_WAIT or TASK_OK, scheduler blocks or releases this task, and then continues the next scheduling cycle.

```
void Scheduler()
  while(1)
    MoveTaskToRun();
    runTask = SelectTaskForProc();
    returnValue = runTask.pFun(runTask);

    if returnValue == TASK_OK
      DeleteTask(runTask);
    else                    // return TASK_WAIT
      AddToQueue(runTask, waitingQueue);
    end;
  end;
```

Fig. 3 Scheduling algorithm

III. CANOPEN MASTER

A. Processors for Received Messages

CANopen master uses interruption request (IRQ) function to receive messages. Part of the flow is presented in Fig. 4. Items with bold circle are tasks, while items with thin circle are data structures.

CANopen has three doubly linked receiving message queues, RPDO, RSDO and RNMT queues. Non-empty of these queues will awake their distributors who process the messages.

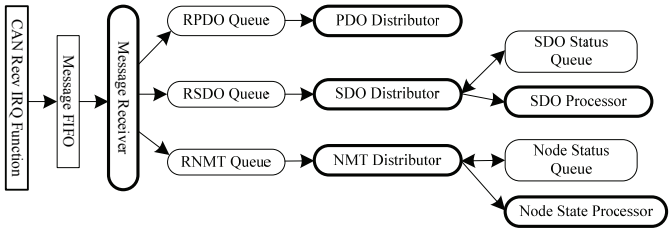

Fig. 4 Process flow of receiving messages

Awaked by non-empty of RPDO queue, PDO distributor analyzes all RPDO messages. To each PDO message, PDO distributor compares its communication object identifier (COB-ID) and that of all RPDO parameters stored in OD. If there is a match, data in this RPDO are directly mapped into OD according to mapping parameters.

SDOs are slightly different. One SDO communication comprises of a sequence of SDO messages. For each sequence, CANopen master has an SDO status, in SDO status queue to record sequence status, and an individual SDO processor processing messages belong to this sequence. Therefore, SDO distributor scans all nodes in SDO status queue to find an existed sequence, and awakes corresponding SDO processor. If no match in this search, distributor scans client SDO parameters in OD, and generates new SDO status and SDO processor. After all, distributor deletes this SDO message under the condition that no match can be found in SDO status queue and client SDO parameters.

As the NMT master in network, CANopen master is in charge of the slave node status tracing [2]. During initial process, master builds a node status in node status queue for each slave known in network. When receiving a NMT message, NMT distributor scans node status queue to find a match, and awake corresponding node status processor. In case the message comes from a new slave, distributor dynamically generates node status and status processor to boot up this node and keep tracing.

Considering the procedure of receiving messages from IRQ functions and inserting them into three receiving message queues, atomic process is an important issue. Since IRQ stops CANopen in an uncontrollable manner, and deleting or inserting a node in doubly linked list is not an atomic operation, directly inserting messages into receiving queues by IRQ functions could damage the queue structure. Therefore, procedure of message receiving and queue inserting are divided into two parts. IRQ function only receives messages from CAN controller. Message receiver task analyzes these messages and inserts them into receiving queues. IRQ function and message receiver are connected by a message FIFO.

B. Processors to Transmit Messages

CANopen master sends five kinds of messages: SYNC, acyclic TPDO, cycled TPDO, TSDO and TNMT messages. Specially, only during synchronous window can cycled TPDO be transmitted, while acyclic PDO is transmitted whenever an event happens. Master only send TNMT message requesting node status under node guard protocol. Indicated by different COB-ID groups, messages are priority scaled. In particular, SYNC message is sent before any other messages, and remote requests for node status can only be sent when no other messages are waiting. There are five transmitting queues in CANopen master, depicted in Fig. 5.

Non-empty of any transmitting queue awakes message sender task. According the underlying priories, message sender guarantees messages of lower priority waiting in queue before transmission of messages in higher priority queues. However, the only exception is cycled TPDOs. Outside synchronous window, message sender does not check the cycled TPDO queue. As cycled TPDOs may stay in queue, SYNC producer task awakes message sender every communication period, ensuring the transmission of cycled TPDOs left in queue. Meanwhile, message sender uses

transmission IRQ. IRQ function always awakes message sender after a transmission until empty of all five transmitting queues.

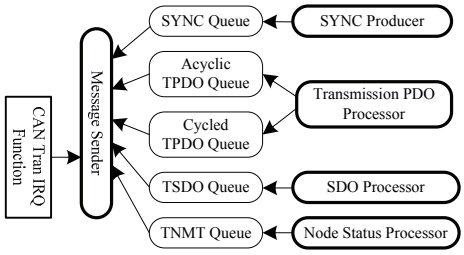

Fig. 5 Flow of sending messages

C. SYNC producer

SYNC producer generates SYNC message periodically. Fig. 6 explains the algorithm of SYNCProducer().

```
char SYNCProducer(TaskObj)
  state = TaskObj.Argu[0];          // read current state
  switch(state)
    case INIT:                      // initial parameters
      cobID = GetODEntry(0x1005,0);
      syncPeriod = GetODEntry(0x1006,0);
      syncWin = GetODEntry(0x1007,0);
      if(syncPeriod > 0)            // SYNC enabled
        SetWait(syncPeriod);        // wait for a SYNC period
        TaskObj.Argu[0] = GEN_SYNC;
      else                          // SYNC disabled
        SetWait(INFINITE);          // block task forever
      end;
      return TASK_WAIT;

    case GEN_SYNC:                  // generate SYNC
      sendSYNC();                   // send a SYNC message
      Global.SYNCWinFlag = 1;       // set global flag
      TriggerPDO();                 // update SYNC counters
      TriggerMsgSender();           // awake message sender
      if (syncWin<syncPeriod) && (syncWin>0)
        SetWait(syncWin);           // wait a sync period
        TaskObj.Argu[0] = WAIT_WIN;
      else                          // sync window disabled
        SetWait(syncPeriod );       // wait a sync window
        TaskObj.Argu[0] = GEN_SYNC;
      end;
      return TASK_WAIT;

    case WAIT_WIN:                  // sync window over
      Global.SYNCWinFlag = 0;       // reset global flag
      SetWait(syncPeriod – syncWin);  // wait to next sync
      TaskObj.Argu[0] = GEN_SYNC;
      return TASK_WAIT;
  end;
```

Fig. 6 Algorithm of SYNC producer task

OD entry 1005h defines the COB-ID of SYNC message, 1006h defines communication period and 1007h defines synchronous window length. Modifications of OD entries 1005h~1007h reset SYNCProducer to state INIT and make it to refresh COB-ID, communication period and synchronous window length. Then, SYNCProducer transfers to state GEN_SYNC and enter its normal procedure. During state GEN_SYNC, SYNCProducer sends out SYNC message and sets global flag SYNCWinFlag. If configuration of synchronous window is valid, SYNCProducer enters state WAIT_WIN to reset SYNCWinFlag, which keeps message sender from sending cycled TPDOs.

D. Transmission PDO processor

Although received cycled and acyclic PDOs are processed equally, transmissions are different.

Every TPDO has a TPDO processor, generated after configuration of PDO pair. In Fig. 6, SYNCProducer calls TriggerPDO(), increasing SYNC counter, and waking up cycled TPDO processors every communication period. Thus, cycled TPDO processors run every communication period with updated SYNC counter. When this value equal with or bigger than PDO transmission type in operational mode, TPDO processor assembles a new PDO message and send it.

However, acyclic TPDO is another case. According to definitions, a modification in OD entry mapped into an acyclic TPDO immediately results in a new TPDO transmission [1]. Unfortunately, TPDO mapping is a one-direction relationship. From an OD entry, CANopen master cannot tell the TPDO mapping the entry without searching all TPDO mapping parameters. To avoid this time consuming search, a new field is added into OD entry to record the PDO number if mapped into an acyclic PDO. Configuration of TPDO is responsible for initializing this field. Hence, when acyclic TPDO mapped OD entry changes, CANopen master can directly identify and awake the related TPDO processor by the PDO number stored. As a sword has two blades, any modifications of PDO mapping and transmission type should clear the PDO number field before set new values, avoiding error triggers to TPDO processors.

E. SDO processor

SDO communication is the major method to configure OD entries in slave nodes. Numerous events launch SDO communications, such as commands from high-level software, modifications in OD, or node boot-up sequences.

To launch an SDO communication, a new SDO processor and an SDO status are generated and initialized with communication parameters. SDO processor generates TSDO messages, uploading or downloading SDO entries, and analyzes RSDO messages to check the communication results. SDO status stores the sequence of SDO communication and helps SDO distributor to find corresponding SDO processor.

When the SDO communication finished, SDO processor is responsible for releasing the SDO status, and writing back result of this communication to the task which launched this communication.

F. Network management

As shown in Fig. 4, every slave in network has its own node status in node status queue, which records node mode and its node state processor. According to the selection between heart beat and node guard protocol, node state processor periodically checks the heart beat messages received or sends out remote request for node statuses.

When CANopen master enters pre-operational mode from power up, all slaves indicated as mandatory node by 1F81h in OD [2] are booted up by a sequence shown in Fig. 7. Optional nodes are also tried with this sequence but network can enter operational mode without them. During operational mode, arrival of boot-up message also launches this sequence.

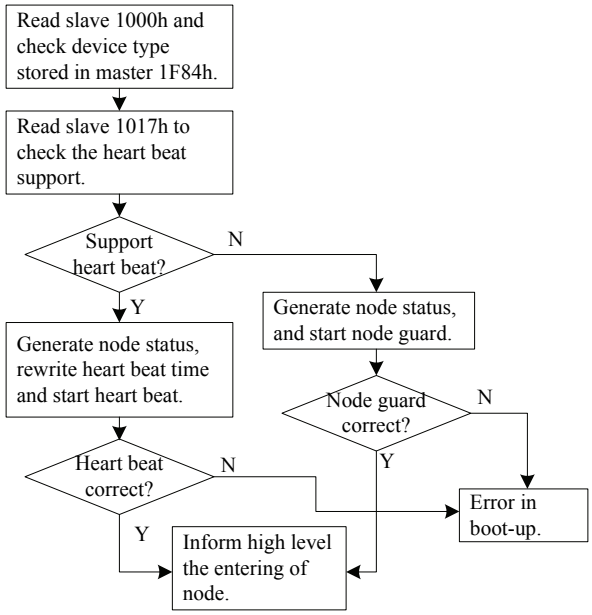

Fig. 7 Simplified boot-up check sequence

However, this is a simplified boot up sequence compared to that defined in [2]. Functions like software version checking and program downloading are not implemented, currently unnecessary in HEV system.

IV. Performance

A. Capability for Transplanting

Although major part of CANopen master is coded in ANSI C, three parts require modification during transplanting: driver for CAN controller, implementation of a microsecond timer, and interface with high-level software.

On operating system supported platform, driver for CAN controller, task scheduler, and high-level software run as three threads. Priority of scheduler is higher than high level software but lower than driver. Tasks for CANopen are scheduled as sub thread in a system level thread. However, the scheduling algorithm of operating system could compromise real-time performance of CANopen. Scheduler should try to sleep at preferred uncritical time, reducing the probability of being unexpectedly blocked by operating system. Therefore, function MoveTaskToRun() in Fig. 3 is modified to return the maximal allowable sleep time. When runnable queue is empty, scheduler is allowable to sleep this time without performance degradation.

When no operating system, driver for CAN controller runs in IRQ function as mentioned. However, the high-level software may run along with CANopen master in the form of task or on other processors by communication of UART of ether net. Therefore, interface with high-level software is a communication interface.

B. Real-time Performance

Latency from releasing last task to launching new task is scheduling time. Test results of scheduling time under different platforms are described in Table II.

TABLE II
SCHEDULING TIME PERFORMANCE

Platform	Clock Frequency	Average Scheduling Time
Pentium 4	3.0GHz	2.28μs
ARM7 LPC2294	44.2368MHz	12.63μs

As a non-preemptive scheduler, scheduling time cannot fully define response time. As mentioned, the highest exchanging rate in HEV system is once per 500 microseconds; therefore, the maximal number of receiving PDO processed in a communication period determines the real-time performance. Shown in Fig.8, even on ARM7 with the 44.2368MHz main clock, CANopen master can analyze and map five receiving PDOs in one communication period. In fact, transmission time for five PDOs on CAN bus may already exceed 500 microseconds, and system is unlikely to require the updates of all PDOs every communication period. Therefore, CANopen master meets the timing requirements of HEV control system.

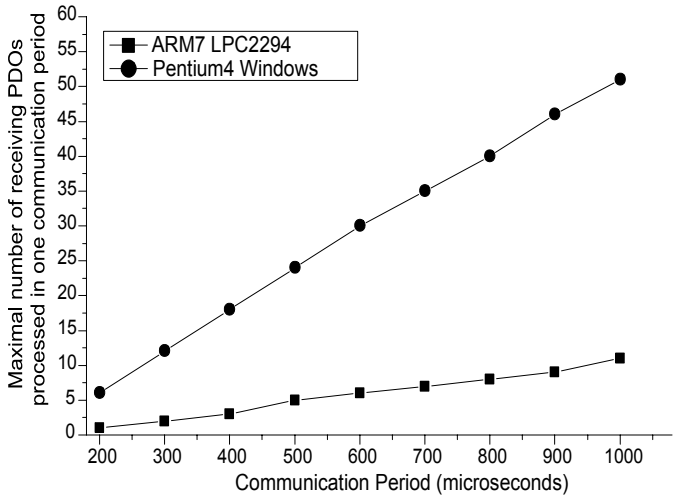

Fig.8 RPDO processed in one communication period

C. Standard Conformance

Table III provides the detailed conformance. Since the use of task scheduler, there is no limitation for concurrent SDO communications and PDO pairs. All CANopen standards required functions are capable to be implemented in this master, except for the electronic data sheet (EDS) storage related functions, which call for support of file system. However, the EDS storage is not necessary for embedded control system.

TABLE III
STANDARD CONFORMANCE

Function	Description
Node type	Master node only.
SYNC	Producer.
SDO	Unlimited SDO sever and client; normal, expedited download and upload support.
PDO	Unlimited PDO receive and transmit; remote request, cycled and acyclic PDO support.
NMT	Heart beat and node guard support. Automatic boot up check.
Time	Time producer.
Emergency	Directly inform high level.

V. CONCLUSION

In this paper, by analyzing the behaviors of CANopen network, we maintain that real-time and concurrent processing along with dynamically run-time adjustment capability are basic features of CANopen master. Apparently, task scheduler is a feasible solution. To provide transplanting capability, we propose a non-preemptive ANSI C task scheduler. This task scheduler merely implements CANopen related items, utilizes the scaled runnable queue from Linux kernel, and realizes a microsecond level non-preemptive O(n) scheduling algorithm.

Based on this scheduler, we build up a standard compatible CANopen master which supports unlimited SDO communication and PDO pairs. Experimental result proves that this CANopen master meets the timing requirement of HEV control system.

ACKNOWLEDGMENT

Authors would like to thank Professor Jianmin Duan who initialized this research, Mingjie Zhang, Zhuo Zhang, Jinjun Xiao and Tao Chen who participated into the design work.

REFERENCES

[1] CAN in Automation e. V., *CANopen - Application Layer and Communication Profile*, CiA Draft Standard 301, Version 4.0.2, February 2002.

[2] CAN in Automation e. V., *CANopen - Framework for CANopen Managers and Programmable CANopen Devices*, CiA Draft Standard Proposal 302, Version 3.1.2, June 2002.

[3] Renji V Chacko, Dr. Z V Lakaparampil, and Chandrasekar.V, "CAN based distributed real time controller implementation for hybrid electric vehicle," *2005 IEEE Conference on Vehicle Power and Propulsion*, pp. 247-251, September 2005.

[4] Junbo W, Bugong X, and Qingyang W, "Analysis and optimization of message scheduling based on the CANopen protocol," *Proceedings of the 25th Chinese Control Conference*, pp. 1815-1819, August 2006.

[5] G. Cena and A. Valenzano, "Efficient polling of devices in CANopen networks," *Proceedings of the ETFA03 IEEE Conference*, pp. 123-130, September 2003.

[6] Krishna, C.M. and Yann-Hang Lee, "Scanning the issue - special issue on real-time systems," *Proceedings of the IEEE*, vol. 91, pp. 983-985, July 2003.

[7] S. Molloy and P. Honeyman, "Scalable linux scheduling," *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 285-296, June 2001.

[8] R. Love. *Linux Kernel Development*, Sams Publishing, 2004.

[9] Shlomi Dolev and Alexander Keizelman, "Non-preemptive teal-time scheduling of multimedia tasks," *Real-Time Systems*, vol. 17, pp. 23-29, July 1999.

[10] Kevin Jeffay, Donald F. Stanat, Charles U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, pp.129-139, December 1991.

[11] Aloysius K. Mok, Wing-Chi Poon, "Non-preemptive robustness under reduced system load," *2005 Real-Time Systems Symposium*, pp. 1-10, December 2005.

[12] Robert Love, "Kernel korner: the new work queue interface in the 2.6 kernel," *Linux Journal*, vol. 2003, pp. 9, November 2003

[13] Jianmin Duan, Jinjun Xiao, and Mingjie Zhang, "Framework of CANopen protocol for a hybrid electric vehicle," *2007 IEEE Intelligent Vehicles Symposium*, in press.

[14] Olaf Pfeiffer, Andrew Ayre, and Christian Keydel, *Embedded Networking with CAN and CANopen*, RTC Books, San Clemente, CA, 2003.

[15] J. Roberson, "UlE: A modern scheduler for FreeBSD," *In Proceedings of BSDCon'03*, pp. 17–28, September 2003.

[16] K.M. Zuberi and K.G. Shin, "Non-preemptive scheduling of messages on controller area network for real-time control applications," *Proc. Real-Time Technology and Applications Symposium*, pp. 240-249, May 1995.

[17] Laurent George, Nicolas Rivierre, Marco Spuri, "Preemptive and non-preemptive real-time uni-processor scheduling," Tech. Rep. RR-2966, INRIA: Institut National de Recherche en Informatique et en Automatique, 1996.