INASS

International Journal of
Intelligent Engineering & Systems

http://www.inass.org/

Design of Embedded WEB Remote Monitoring System Based on μC/OS-II Operating System

Xibo Wang, Zhen Zhang*

School of Information Science and Engineering, Shenyang University of Technology, Shenyang 110870, China

* *Corresponding authors' Email: zhangzhen0370@126.com*

Abstract: Isolating embedded system has not been adapted to current development tendency. More and more embedded systems need networking for communicating with each other. μC/OS-II, which is a typical embedded real time operating system and contains a kernel without network protocol, has been widely used in a large number of embedded systems. So it is of significance to join network protocol into μC/OS-II. In this paper, a networking method is proposed to embedded equipments based on μC/OS-II. LPC2200 based on ARM7TDMI-S is used as the core of hardware platform, μC/OS-II embedded real-time operating system and LwIP (Light Weight IP) are used as software platform, ADS1.2 as compiler. Through the realization of files that are related to the processor LPC2200, the porting of μC/OS-II into ARM7 is implemented. Through the realization of the LwIP operating system emulation layer and realization network chip RTL8019AS driver, the porting of LwIP into μC/OS-II is implemented. On these foundations, a remote monitoring application is successfully developed. Experiment results show that proposed approach can be successfully used for embedded equipments networking control.

Keywords: embedded system; LwIP; μC/OS-II.

1. Introduction

Embedded system [1] has been widely used in various fields. The import of embedded operating system can enable embedded systems to administer all kinds of resources, run system correctly and steadily, provide bottom interfaces to upper soft, and be convenient for secondary development.

In recent years, with the rapid development of communications technology and internet and progress of embedded technology, embedded system has a great influence on human, and changes human's future life style. But the isolating embedded system has not been adapted to current development tendency. Since more and more needs such as "remote monitoring" [2] and "smart home appliances" [3] are put forward, embedded devices should not only have the traditional control, monitor, auxiliary equipment, machinery and equipment operation function, but also be connected to the Internet, make full use of network resources to realize remote monitoring, remote control and wider range of information sharing and intelligent data processing functions. So the use of operating system and network protocol in embedded systems is of vital and practical significance.

μC/OS-II [4] is a multitask real time operating system with open source code which is written by Jean J. Labrosse. μC/OS-II is a high-performance kernel and its instantaneity is guaranteed. μC/OS-II operating system reduces the complexity of program and makes the design and maintenance of application system simple [5]. μC/OS-II operating system can be used in the 8-bit processors, the 16-bit processors and the 32-bit processors. The functions of μC/OS-II op-

erating system can be tailored according to different applications. μC/OS-II operating system demands for low hardware. Its scheduling mode is preemptive i.e. Task whose state is ready and has highest priority can be run [6]. It has implemented task scheduling, task to task communication, management of system memory, but it has no file system and network protocol.

LwIP is a realization of TCP/IP protocol stack. Its goal is to reduce usage rates of memory and the size of code, so LwIP can be applied to this platform such as embedded system whose resource is limited. In order to simplify the process and reduce memory requirement, LwIP has clipping API and in this way LwIP is a high-performance network protocol.

This paper introduces a method that joins LwIP network protocol stack into μC/OS-II. Thus embedded system can access the Internet. On this basis, a simple WEB server is implemented. So users can access and control the development board via the browser.

2. The Overview of μC/OS-II

μC/OS-II is a multitask real-time operating system which is based on preemptive kernel [7, 8]. μC/OS-II is a real time operation system (RTOS) indeed, which is very suitable for intelligent control on site [9]. μC/OS-II has no file system, no network system, no interface system, no peripherals management system, etc. Because the source code of μC/OS-II is clipping, open, portable, it is very suitable to be used as instrument inline micro controller [10].

μC/OS-II uses ANSI C language, contains a small part assembly code. This enables it to be used for microprocessors of different structures [11]. The software structure of μC/OS-II contains three parts code. The first part is core code which is independent of the processor; the second part is header file which is associated with the application configuration; the third part is the code which is dependent of the processor. When porting, it is necessary to write the code which is dependent of the processor. The software structure of μC/OS-II is shown in Figure 1.

3. The Overview of LwIP

TCP/IP protocol is complex. Its implementation is difficult for the embedded devices which lack the powerful support of operating systems. For a particular application, embedded devices often only need a small part of TCP/IP protocol. According to particular request, we use an existing simplified TCP/IP protocol-LwIP, and recompose TCP/IP protocol based on LwIP

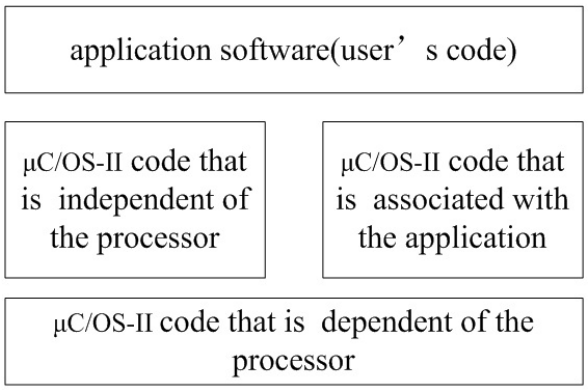

Figure 1 The software structure of μC/OS-II.

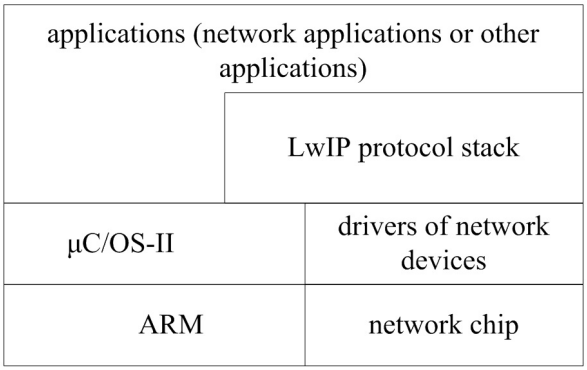

Figure 2 Diagram of the μC/OS-II and LwIP.

[12]. LwIP is an open-source embedded TCP/IP protocol stack. Its name in English is “Light Weight IP” - lightweight TCP/IP protocol stack. Its purpose is to reduce the usage of memory and the size of code and to make LwIP suitable for the embedded system whose resource is very strained. In order to reduce processing program and internal demand, LwIP uses tailored API without data replication [13].

Referring to hierarchical protocol, LwIP has designed and implemented the TCP/IP protocol stack. LwIP is composed of several modules, including the implementation modules of TCP/IP(IP, ICMP, UDP and TCP), operating system emulation layer, buffer and memory management subsystem, network interface functions and a group of Internet verifying and calculating functions. At the same time, LwIP also provides a set of abstract API interfaces. Thereby it is greatly convenient for the developers to develop applications. After putting LwIP in μC/OS-II operating system, the whole system is shown in Figure 2.

4. The Porting of μC/OS-II

The so-called porting means enabling a real time kernel to run on certain microprocessor or microcon-

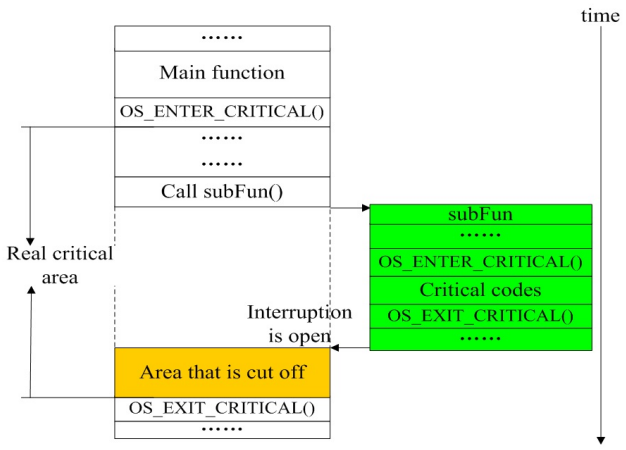

Figure 3 The phenomenon of critical area is cut off.

Table 1 List of functions

Function	Must be realized
OSTaskStkInit()	Yes
OSInitHookBegin()	No
OSInitHookEnd()	No
OSTaskCreateHook()	No
OSTaskDelHook()	No
OSTaskIdleHook()	No
OSTaskStatHook()	No
OSTaskSwHook()	No
OSTCBInitHook()	No
OSTimeTickHook()	No

troller [14]. According to the porting request of μC/OS-II, the process of porting μC/OS-II to a new system structure needs 3 files: "OS_CPU.H" (C header file), "OS_CPU_C.C" (C language program source file) and "OS_CPU_A.S" (assembler source file).

"OS_CPU.h" uses "typedef" to define some data types which are connected with processor, uses C preprocessor directive "#define" to define some constants and macros which are also connected with processor. Because ADS1.2 only supports full stack decrease progressively, the value of the constant "OS_STK_GROWTH" that decides growth direction of the stack must be "1". After defining macro "OS_CPU_EXT", we can use "OS_CPU_EXT" to define a global variable. Because the word length of different CPUs might be different, so the data types of C language must be redefined.

Towards the processing program of critical codes, because the use of the first kind of interrupt mode may produce a status in the critical area of main function, sub function that includes critical area is called. Before the return of sub function the interrupt has been opened, then it will lead to the interrupt having been opened before the critical area of main function has run completely. This phenomenon is shown in Figure 3. So the first interrupt mode can not be used. For the second interrupt mode, critical area code has modified stack pointer that is used to save interrupt state, thus it may cause an error when interrupt open/close state is resumed, therefore the second interrupt mode can not be used. The open/close interrupt way of μC/OS-II is defined as "3", the interrupt states are saved as local variables, so this method can avoid the two wrong methods what we said above.

Macro "OS_ENTER_CRITICAL()" and "OS_EXIT_CRITICAL()" are defined and used to open/close interrupt. In this file, some function prototypes are declared, so that other files can use them expediently. These functions include OSTickISR(), OSCtxSw(), OSIntCtxSw(), OSStartHighRdy(), ARMCoreDisalbeItExt(),ARMCoreRestoreIntStatus(). These functions will be realized in file "OS_CPU_A.s".

In "OS_CPU_C.c" file, there are two group functions. The first group functions must be realized, and the second group function can be realized as empty functions. These functions are shown in Table 1. From Table 1, we can see there is only one function that must be realized. The other functions are hook functions. These hook functions can be put into the kernel of μC/OS-II, then the function of kernel can be extended. But there is no need to extend the function of kernel in this paper. These hook functions are only defined.

μC/OS-II is a multitasking operating system, the task will not run immediately after it has been created, it must wait for the kernel scheduler program which has pushed it into the CPU registers from stack, then it runs. So every task stack must always keep right data and the data is enough to make the task run or recover running state. This theory is shown in Figure 4. So the stack of task must be properly initialized when a task is created, in other words, function "OSTaskStkInit()" must be realized in file "OS_CPU_C.c". When a task is created, the function "OSTaskStkInit()" is called. Function "OSTaskStkInit()" realizes the stack of task shown in Figure 4. Task enter address is saved in PC register in the stack of task. When scheduler program puts task enter address into PC register in CPU, CPU will run this task. Some other user functions are called when μC/OS-II do some operation defined as empty functions.

μC/OS-II is a deprivable kernel, before it starts, the

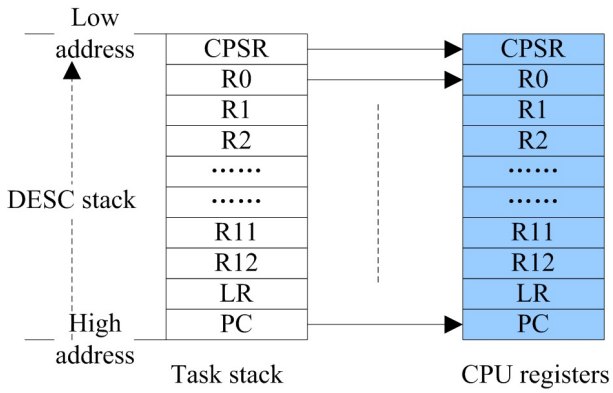

Figure 4 The relationship Between task stack and CPU register group

CPU should be owned by the highest priority task whose status is ready. Function "OSStart()" looks up task which should be run first. After function "OSStart()" finds one task, function "OSStartHighRdy()" will put stack of this task into stack of CPU. Because this belongs to operation of registers, it belongs to the porting of μC/OS-II. It needs to realize code in line with different CPUs. The only work that function "OSStartHigRdy()" should do is to put stack of task, whose priority is the highest and whose status is ready, into stack of CPU, and recover task scene that function "OSTaskStkInit()" simulates into stack of CPU, then the task can run.

When interrupt returns, function "OSIntCtxSw()" is used to switch tasks. After ISR finishes the treatment of interruption, μC/OS-II has to make a task schedule. The aim is to ensure that interrupt handling results can get timely response. So the all interruptions of μC/OS-II must have one and only entry and exit. In order to achieve this goal, exception vectors must be modified. IRQ and FIQ point to a uniform entry function. After user-defined interruption is disposed completely, μC/OS-II makes a task schedule. The process of function "OSIntCtxSw()" is shown in Figure 5. In Figure 5, we can see that variable "OSPrioCur", where priority of current task is saved, is revised as priority of task being about to run. Variable "OSTCBCur", where address of current task OS_TCB is saved, will be revised as the address of task being about to run. The stack pointer register (SP_svc) points to the top of stack of task being about to run. CPSR register pops from the stack. AIC_EOICR register is written. This notices that the interruption is end. The task that being about to run pops from stack. Then the task can run.

Function "OSTickISR()" is used for interrupt service routine of clock tick. "OSTickISR()" must be called by μC/OS-II kernel ISR. "OSTickISR()" reduce "OSTCBDly" of every task.

Function "OSSched()" needs function "OS_TASK_SW()" to complete task switching work. Because task switching needs to operate CPU registers, and different CPUs have different operation instructions to operate registers, function "OS_TASK_SW()" is realized according to different CPUs in "OS_CPU_A.s" file. The process of function "OS_TASK_SW()" is shown in Figure 6. From Figure 6, we can see that running scene of current task is saved into stack of task, the pointer of current task is saved into OS_TCB of current task, variable "OSPrioCur", where priority of current task is saved, is revised as priority of task being about to run, variable "OSTCBCur" where address of current task OS_TCB is revised as address of task being about to run, the pointer of task being about to run is recovered from OS_TCB to SP register, the scene of task being about to run is recovered into CPU register, then the task being about to run can run.

5. The Porting of LwIP

The porting of LwIP is divided into two parts: one part is to realize the operating system emulation layer, i.e. "interface layer"; another part is to realize bottom-level driver. The operating system emulation layer provides a common interface between the LwIP code and the underlying operating system kernel. The general idea is that porting LwIP to new architectures requires only small changes to a few header file. In order to facilitate porting of LwIP, functions and data structure that belongs to operating system are not used directly, but an interface layer is used to take the place of these functions. Therefore, in order to the porting of complete LwIP, the interface layer must be realized [15]. So interface layer should be modified according to a given operating system. Because different network card chip has different operation, bottom-level driver should be realized according to different hardware.

Environment variables and data types are defined in file "cc.h". These common data types are used in operating system emulation interface functions and low-level protocol stack. Towards the critical-protecting functions of LwIP, LwIP and μC/OS-II have the same way to handle this, so same method can be used to realize critical-area-protecting functions as μC/OS-II. Macro "SYS_ARCH_PROTECT()" and "SYS_ARCH_UNPROTECT()" are defined and used to realize the protection of critical area.

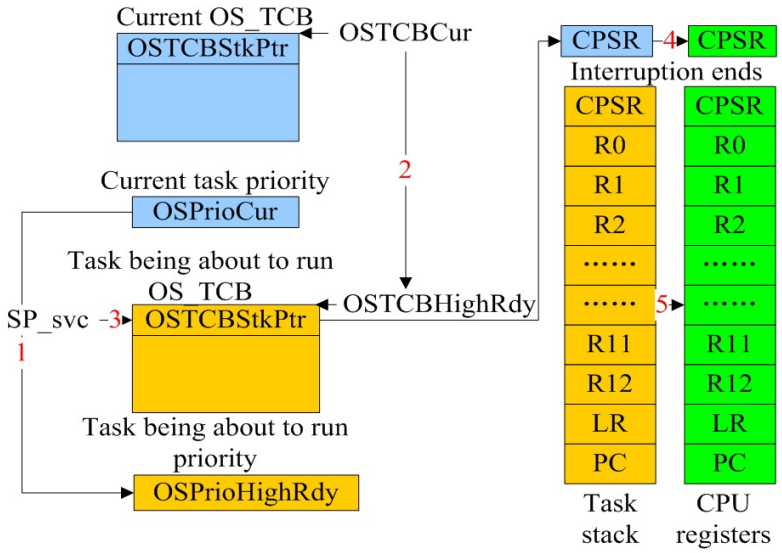

Figure 5 The task switching process at interruption level

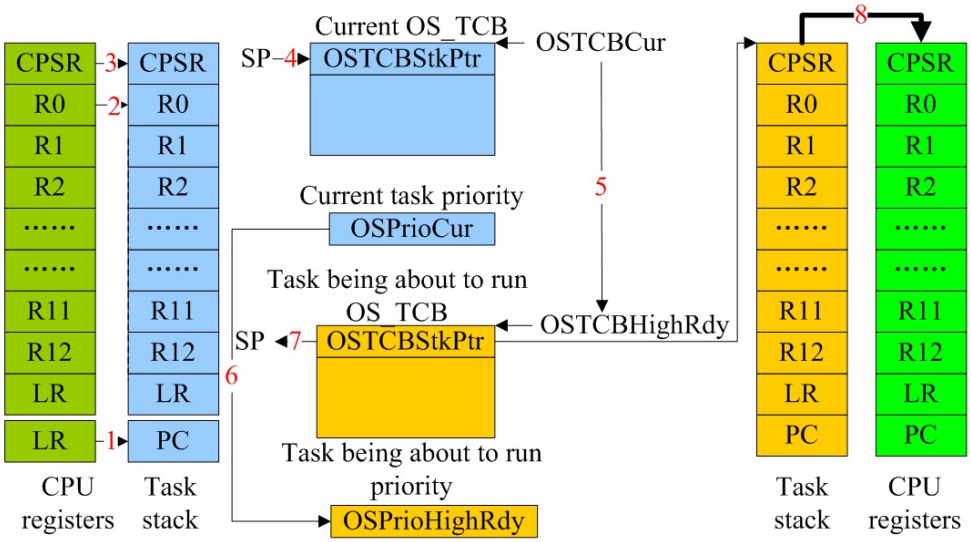

Figure 6 The task switching process at task level

Because ARM7 aligns as 4 bytes as default, if structure designed can not align as 4 bytes, then compiler "ADS1.2" will readjust structure and make sure the structure aligns as 4 bytes, because the visit efficiency is the highest when the structure aligns as 4 bytes. But sometimes structure needs to be visited as actual member boundary, especially in network structures. LwIP has thought about this problem and provided some macros to package structure, so these macros need to be defined. These macros include
"PACK_STRUCT_FIELD(x)",
"PACK_STRUCT_STRUCT",
"PACK_STRUCT_BEGIN", and
"PACK_STRUCT_END".

Semaphores are represented by the type "sys_sem_t" which is defined in the "sys_arch.h" file. Mailboxes are equivalently represented by the type "sys_mbox_t". LwIP does not place any restrictions on how "sys_sem_t" or "sys_mbox_t" are represented internally.

Macro used for performance measurement is defined as an empty macro in file "perf.h".

The following 4 groups of functions must be implemented by the operating system emulation layer.

(1) Semaphore operation functions

LwIP needs semaphores to achieve communication within processes, so semaphore processing functions need to be realized. Semaphores can be either counting or binary - LwIP works with both kinds. These functions are shown as follows.

1) sys_sem_t sys_sem_new(u8_t count)

This function creates and returns a new semaphore. The "count" argument specifies the initial state of the semaphore.

2) void sys_sem_free(sys_sem_t sem)

This function deallocates a semaphore.

3) void sys_sem_signal(sys_sem_t sem)

This function signals a semaphore.

4) u32_t sys_arch_sem_wait(sys_sem_t sem, u32_t timeout)

This function blocks the thread while waiting for the semaphore to be signaled. If the "timeout" argument is non-zero, the thread should only be blocked for the specified time (measured in milliseconds).

If the timeout argument is non-zero, the return value is the number of milliseconds spent waiting for the semaphore to be signaled. If the semaphore isn't signaled within the specified time, the return value is "SYS_ARCH_TIMEOUT". If the thread doesn't have to wait for the semaphore (i.e., it is already signaled), the function may return zero.

Because μC/OS-II has already provided semaphore, these functions mentioned above can be realized by packaging or rewriting semaphore operation functions that μC/OS-II has already provided.

(2) Mailbox operation functions

Mailboxes are used for message passing and can be implemented either as a queue which allows multiple messages to be posted to a mailbox, or as a point where only one message can be posted at a time. LwIP works with both kinds, but the former type will be more efficient. A message in a mailbox is just a pointer, nothing more. So mailbox operation functions must be realized. These functions are shown as follows.

1) sys_mbox_t sys_mbox_new(void)

This function creates an empty mailbox.

2) void sys_mbox_free(sys_mbox_t mbox)

This function deallocates a mailbox. If there are messages still present in the mailbox when the mailbox is deallocated, it is an indication of a programming error in LwIP and the developer should be notified.

3) void sys_mbox_post(sys_mbox_t mbox, void *msg)

This function posts the message to the mailbox.

4) u32_t sys_arch_mbox_fetch(sys_mbox_t mbox, void **msg, u32_t timeout)

This function blocks the thread until a message arrives in the mailbox, but does not block the thread longer than "timeout" milliseconds (similar to the "sys_arch_sem_wait()" function). The message argument is a result parameter that is set by the function (i.e., by doing "*msg = ptr"). The message parameter maybe "NULL" to indicate that the message should be dropped.

The return values are the same as for the "sys_arch_sem_wait()" function: Number of milliseconds spent waiting or "SYS_ARCH_TIMEOUT" if there was a timeout.

μC/OS-II has realized the message queue, and it provides abundant function of message queue operation, then it is needed to package these functions for the need of LwIP.

(3) "sys_arch_timeout" function

In LwIP, each thread has a list of timeouts which is represented as a linked list of "sys_timeout" structures. The "sys_timeouts" structure holds a pointer to a linked list of timeouts. This function is called by the LwIP timeout scheduler and can not return a "NULL" value. The function prototype of "sys_arch_timeouts" is shown as follows.

struct sys_timeouts *sys_arch_timeouts(void)

This function returns a pointer to the per-thread "sys_timeouts" structure. In a single thread the operating

system emulation layer implementation, this function will simply return a pointer to a global "sys_timeouts" variable stored in the operating system emulation layer module.

(4) Functions that are related to thread

These functions are shown as follows.

1) sys_thread_t sys_thread_new(void (* thread)(void *arg), void *arg, int prio)

This function starts a new thread with priority "prio" that will begin its execution in the function "thread()". The "arg" argument will be passed as an argument to the "thread()" function. The id of the new thread is returned. Both the id and the priority are system dependent.

For μC/OS-II, a creation of a new thread means a creation of a task. It is needed to package the function "OSTaskCreate()" that μC/OS-II has provided. In the way of arrangement of priority, in order that priority conflict does not occur, initiative thread priority of LwIP i.e. "T_LWIP_THREAD_START_PRIO" is defined.

2) sys_prot_t sys_arch_protect(void)

This optional function does a "fast" critical region protection and returns the previous protection level. This function is only called during very short critical regions. An embedded system which supports ISR-based drivers might want to implement this function by disabling interrupts. Task-based systems might want to implement this by using a mutex or disabling tasking. This function should support recursive calls from the same task or interrupt. In other words, "sys _arch_protect()" could be called while already protected. In that case the return value indicates that it is already protected.

3) void sys_arch_unprotect(sys_prot_t pval)

This optional function does a "fast" set of critical region protection to the value specified by "pval". See the documentation for "sys_arch_protect()" for more information. This function is required for this port supports an operating system.

6. System Initialization for LwIP

A truly complete and generic sequence for initializing the LwIP stack cannot be given because it depends on the build configuration and additional initializations for your runtime environment. But there is a general method. The following functions must be called in turn.

(1) stats_init()

This function clears the structure where runtime statistics are gathered.

(2) sys_init()

The operating system emulation layer is initialized in this function.

(3) mem_init()

This function initializes the dynamic memory heap defined by MEM_SIZE.

(4) memp_init()

This function initializes the memory pools defined by MEMP_NUM_x.

(5) pbuf_init()

This function initializes the pbuf memory pool defined by PBUF_POOL_SIZE.

(6) etharp_init()

This function initializes the ARP table and queue.

(7) ip_init()

This function should be called to handle future changes.

(8) udp_init()

This function clears the UDP PCB list.

(9) tcp_init()

This function clears the TCP PCB list and clears some internal TCP timers.

(10) netif_add(struct netif *netif, struct ip_addr *ipaddr, struct ip_addr *netmask, struct ip_addr *gw, void *state, err_t (* init)(struct netif *netif), err_t (* input)(struct pbuf *p, struct netif *netif))

This function adds your network interface to the "netif" list. This function allocates a "netif "struct and passes a pointer to this structure as the first argument. The "init" function pointer must point to a initialization function for Ethernet "netif" interface. In this paper this function is "ethernetif_init()". "Ethernetif_init()" function is an entry function for the initialization of underlying network interface. The main work of "ethernetif_init()" function is "send-function" (the sending of data is completed by "send-function"). "Send-function" is written in line with specific hardware. "Send-function" is registered to LwIP on data link layer. A semaphore handle is created for the blocking access to network card. MAC address is filled in "ethernetif" stuct. Network card chip is initialized in order to build a physical connection link and a thread for receiving data is created. At last, the timing update for ARP table is opened. The process of "ethernetif_init()" function is shown in figure 7. In fact, "etherneif_init()" function calls a number of functions, these functions is shown in table 2.

(11) netif_set_default(struct netif *netif)

This function registers the default network interface.

(12) netif_set_up(struct netif *netif) When the netif is fully configured, this function must be called.

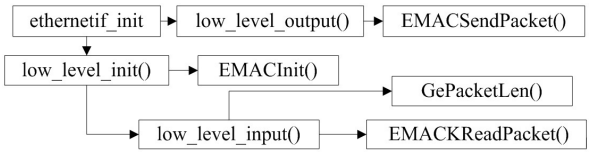

Figure 7 Process of "ethernetif_init()" function.

Table 2 Functions that "etherneif_init" calls

Name	Effect
etherneif_init	initialization entry function for underlying network interface
low_level_output	send-function on data link layer
EMACSendPacket	send frame
low_level_init	network initialization on data link layer
EMACInit	network card chip initialization
GetPacketLen	get length of frame
low_levle_input	receive-function on data link layer
EMACReadPacket	receive frame

(13) dhcp_start(struct netif *netif)

This function creates a new DHCP client for this interface on the first call.

7. Network Device Driver

In this paper, network card chip is RT-L8019AS. RTL8019AS is a highly integrated Ethernet chip, compatible with NE2000 on software level. So the driver of RTL8019AS can be written according to the standard of NE2000. These mainly include initialization, receiving data and sending data.

The initialization function flow chart of RTL8019AS is shown in Figure 8.

The sending function flow chart of RTL8019AS is shown in Figure 9.

The receiving function flow chart of RTL8019AS is shown in Figure 10.

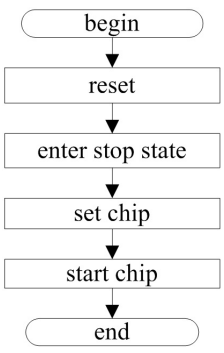

Figure 8 The initialization function flow chart of RTL8019AS.

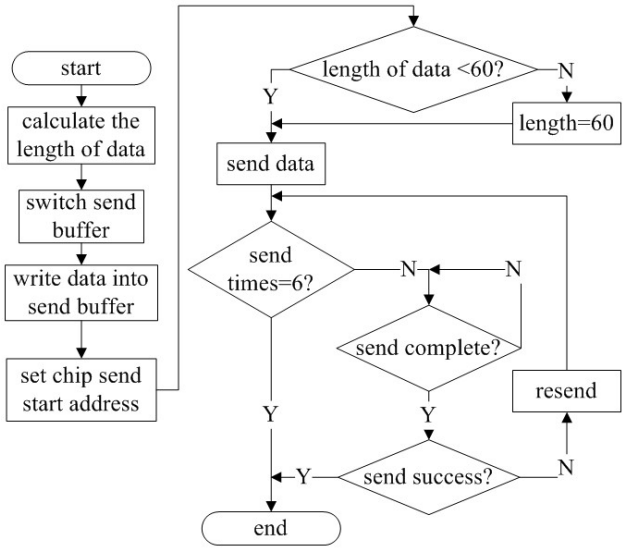

Figure 9 The sending function flow chart of RTL8019AS.

Regarding the interface between network drive and LwIP, the author of LwIP has designed a template. This template is in "/src/netif/ethernetif.c". In LwIP, each network card corresponds with a "netif" structure. LwIP calls function "netif-input" and "netif-output" of "netif" for receiving and sending of Ethernet packets. It is needed to realize these functions that are between upper layer(IP layer) and lower layer(link layer) in "ethernetif.c" file. These functions include "ethernetif_init()" used for initialization of bottom network interface, "low_level_output()" used for sending data on link layer, "low_level_init()" used for initialization of network card, "ethernetif_input()" used for receiving data, "low_level_input()" used for reading a frame.

8. Application design and test

After the porting, application test program can be written. This application is designed to build a WEB server. User can access development board through browser, click on the "submit" button, then the LEDs of the development board are lightened, and LEDs display as water lamp. The main thought is to handle HTTP1.1 protocol. The message that browser submits can be distinguished into "GET" and "POST" request. If a "GET" request arrives, the WEB server sends a default web page to browser. If a "POST" request arrives, according to the function name and parameter that browser submit, the WEB server finds homologous service function, and makes response and deals with the current request. The flow char Figure

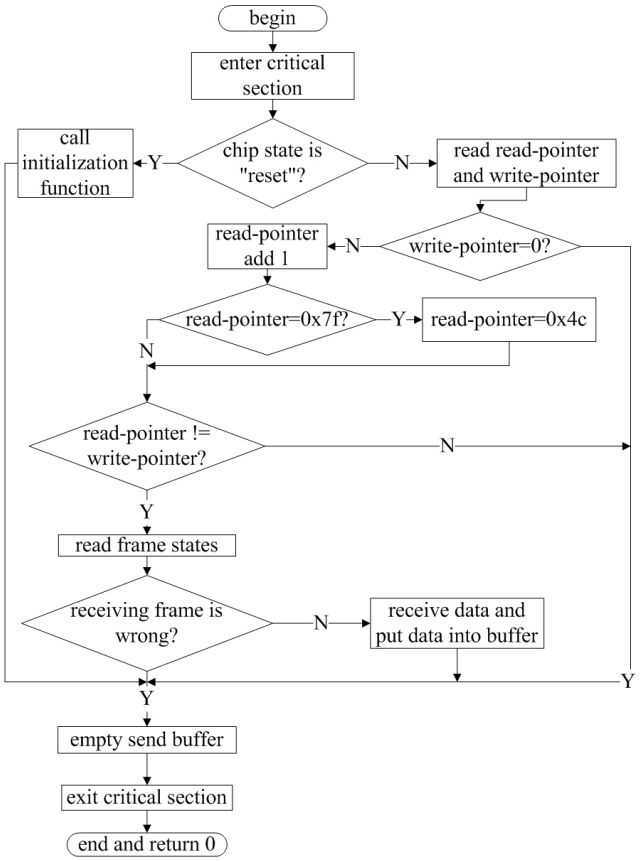

Figure 10 Receiving function flow chart of RTL8019AS.

11 shows the core of application.

After the IP address of development board is input in the address field of browser, a "GET" request is sent to development board. Then development board sends a static Web page to browser. The result of visiting development board in the browser is shown in Figure 12.

After clicking "submit", a "POST" request is sent to development board. Then development board enables water lamp according to the "POST" request. The status of development board is shown in Figure 13.

9. Conclusion

In this paper, embedded light weight network protocol stack (LwIP) is designed and implemented successfully. A simple WEB server is implemented. Looking from the status of development board, the expected result is obtained. TCP/IP protocol is added into μC/OS-II successfully. The μC/OS-II operating system suits for various embedded equipments preferably.

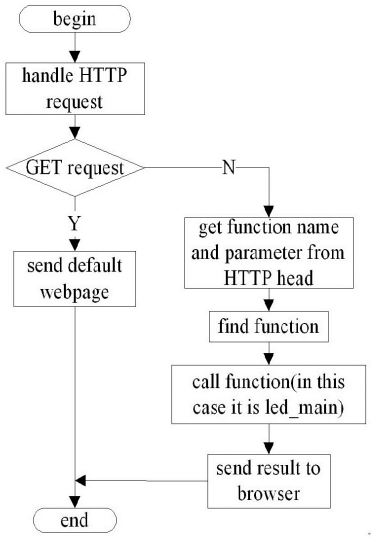

Figure 11 Flow char of the application core.

Figure 12 Result of visiting development board in the browser.

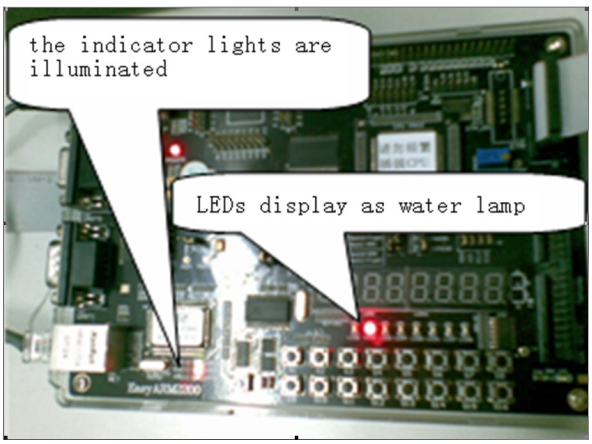

Figure 13 Status of development board.

Acknowledgments

We would like to thank Qiao Zhou for his support and helpful comments.

References

[1] H. Liu, "Simple discussion on the development tendency of embedded system", In: *China Construction Education*, Beijing, China, pp.51-54, 2006.

[2] L. Li, J. He, C. Cao and J. Li, "PLC Remote Monitoring Application Research Based on TCP/IP protocol", In: *Microcomputer Information*, Beijing, China, pp.57-59, 2011.

[3] Z. Chai, "Intelligent Home Systems of Remote Control Based On Internet", In: *Computer Knowledge and Technology*, Hefei, China, pp.7194-7195, 2009.

[4] J. L. Jean, *The Embedded Real-Time Operating System μC/OS-II*, Beijing University of Aeronautics and Astronautics Press, Beijing, 2005.

[5] H. Qin, Y. Tang, "Transplant of Real-time Operating System, μC/OS-II on the HFRK2410C", In: *Computer Knowledge and Technology*, Hefei, China, pp.1544-1547, 2010.

[6] H. Ji, S. Fu, X. Che and L. Zhou, "Analysis and Improvement of Real-time Embedded Operating System μC/OS-II Kernel", In: *Computer Engineering*, Beijing, China, pp.246-247, 2007.

[7] J. Shen, and X. Liu, "Analysis of μC/OS-II Kernel and Realization of Multi-task Schedulers", In: *Computer Engineering*, Beijing, China, pp.85-87, 2006.

[8] Y. Wu, W. Chen, S. Zheng and H. Xu, *Embedded Real-Time Operating System μC/OS- Course*, Xidian University Press, Xi'an,2007.

[9] Y. Liu, L. Cao, "Acceleration and Deceleration Control for Step Motor Base on UC/OS-II", In: *Intelligent Information Technology Application Workshops*, Shanghai, China, pp.180-183, 2008.

[10] L. Zhou,*ARM Embedded System Foundation Course*, Bei Hang University Press, Beijing, 2005.

[11] J. Luo Y. Xie H. Shu and Y. Zhang, "Implementation of LwIP in uC/0S II", In: *Microcomputer Information*, Beijing, China, pp.46-47, 2005.

[12] H. Jiao, *The Design of Embedded Network System*, Bei Hang University Press, Beijing, 2008.

[13] Adam Dunkels, LwIP-1.4.0 source code, 2011.

[14] C. Li, M. Huang, X. Zhang, J. Hu, Y. Liu and Y. Wang, "Transplant Method of μC/OS- on XC164CS", In: *Computer Engineering*, Beijing, China, pp.242-244, 2010.

[15] M. Cheng, Z. Yu, Y. Su and X. Guo, "Porting LwIP into μC/OS- and Testing", In: *Microcomputer Information*, Beijing, China, pp.79-90, 2008.