Network- and Internet Integration of Embedded Systems

Peter Balog
University of Applied Sciences Technikum Wien
A-1200 Wien, Höchstädtplatz 5
balog@technikum-wien.at

Abstract. This paper deals with different approaches and solutions for enabling network connectivity in the embedded systems domain. Furthermore an RTOS-based open software architecture is introduced in order to realize flexible applications with functional and non functional requirements on network and internet connectivity.

The main advantage of this "open solution" in comparison to plug & play solutions is that the desired network functionality can be customized, even if it is necessary to extend or violate some network protocol standards, which is not unusual in embedded system applications due to specific constraints and limitations.

1 Introduction

Today, Internet accessibility in one form or another, if not an a priori requirement, is at least a highly desirable option in many embedded applications. Internet connectivity – or in more general words – network connectivity for embedded systems can be seen as an enabling technology for both current and future applications. Even today's low cost microcontrollers are strong enough to handle a LAN controller (on-chip, off-chip), a communication stack and usually multithreaded applications.

2 Types of applications

Actually there are two types of network applications distinguished by their requirements for network capability. The first category of applications requires network connectivity as a core feature (functional requirement) to establish the desired functionality, e.g. all kinds of mobile computing, health care applications (patient monitoring), home automation and home security, as well as several environment monitoring tasks.

Additional features, e.g. system maintenance, remote configuration, remote system control and monitoring, and software upload, are the motivations for the second class of applications. These so called non-functional requirements are primarily not required for the device's main tasks, but lead to increasing system flexibility.

3 Out of the box solutions

In the following a few out of the box hardware-/software-solutions are introduced:

3.1 LANTRONIX XPort

The XPort is connected to a serial interface of a microcontroller and enables web- and other network-access by means of a fully developed TCP/IP network stack and OS running on a high performance microprocessor with a 10/100Mbit Ethernet interface. Mechanically the XPort is integrated in a 33,9x16,25x13,5 mm^3 RJ45-Package. The microcontroller software application communicates via high level commands (e.g. AT-commands) to the XPort.

3.2 Beck IPC@CHIP

The IPC@CHIP is an embedded controller designed to WEB- or LAN-enable products implemented in a 32 pin 600mil package. The hardware consists of a 16 bit 186 CPU, RAM, Flash, Ethernet, Watchdog and power-fail detection. The preinstalled software consists of a Real Time Operating System (RTOS) with file system, TCP/IP stack, web server, FTP server, Telnet server and Hardware interface layer, providing a high level API to the application programmer. The goal is to put the whole application into this chip.

3.3 High end Microcontroller systems

An embedded system based upon a high end 32-Bit Microcontroller – usually equipped with an on-chip LAN-unit –, such as the *Coldfire* (Freescale Semiconductors), has enough power to run an operating system like *µCLinux* together with *inet*-demon supporting full internet connectivity. The software API for network applications is the *BSD Socket Interface* [3].

3.4 High end Microprocessor systems

An embedded system based upon a high end 32-Bit Microprocessor, such as a *PowerPC*, has enough power to run an operating system like a full *Linux* or *RT-Linux*. The design environment for networking applications is quite the same as for platforms, e.g. Unix-Workstations and PCs.

3.5 Low cost Microcontroller systems

Widespread used 8- and 16-Bit Microcontrollers are generally not equipped with dedicated LAN interfaces but with a variety of serial and parallel IO units. To obtain network and internet connectivity additional hardware (and software, of course) has to be added.

4 Principle approaches

Basically there are three suitable approaches embedded systems engineers can apply in order to achieve internet connectivity for the embedded device:

4.1 Internet connectivity via a modem facility

By means of a standard serial interface (e.g. RS-232) a communication path to a modem-chip or modem-device (fig. 1) is established. The other side of the modem is connected either to a POT-, ISDN-, or xDSL-line. A TCP/IP-stack with a serial network layer (SLIP, PPP) has to be implemented into the systems software or firmware of the embedded device.

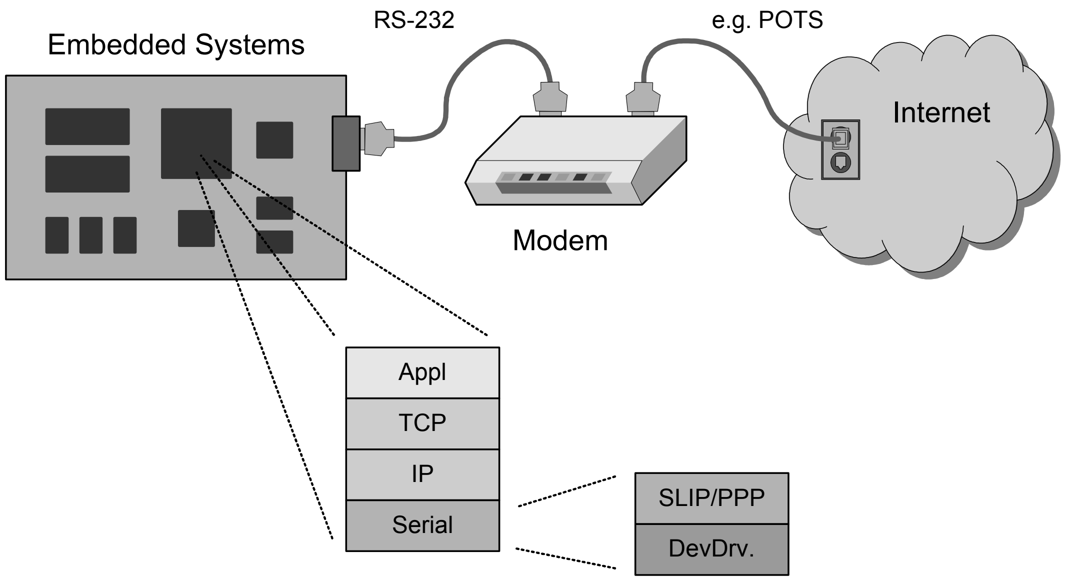

Fig 1: Internet connectivity via Modem

4.2 Using a Gateway-Computer

By means of a standard serial interface (e.g. RS-232) a communication to a *gateway computer* is established. The *gateway computer* itself is equipped with a standard LAN/Internet interface; the entire internet related software (TCP/IP-stack, Web server with CGI-functionality [2]) runs on this *gateway computer*. There's no need for any standard protocol between the embedded system and the gateway.

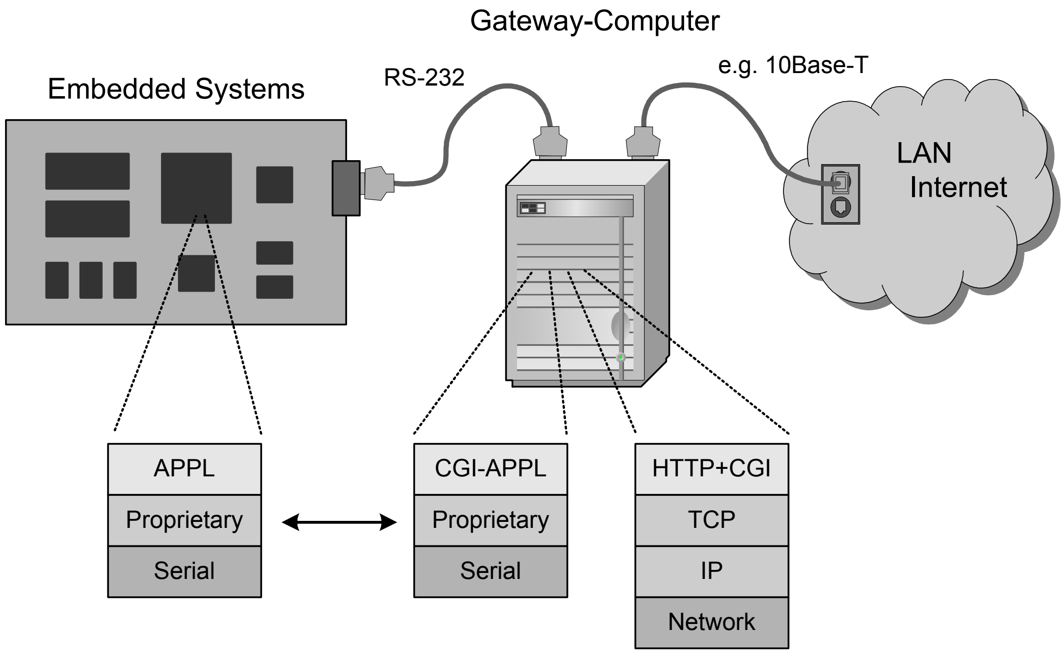

Fig 2: Internet connectivity via a gateway computer

4.3 Direct LAN access

The embedded system hardware is equipped with a LAN controller to access a network directly. The entire networking and internet related software (LAN-driver, TCP/IP-stack including all application modules like web-, ftp-, and telnet-server), as well as the application software run on the embedded system.

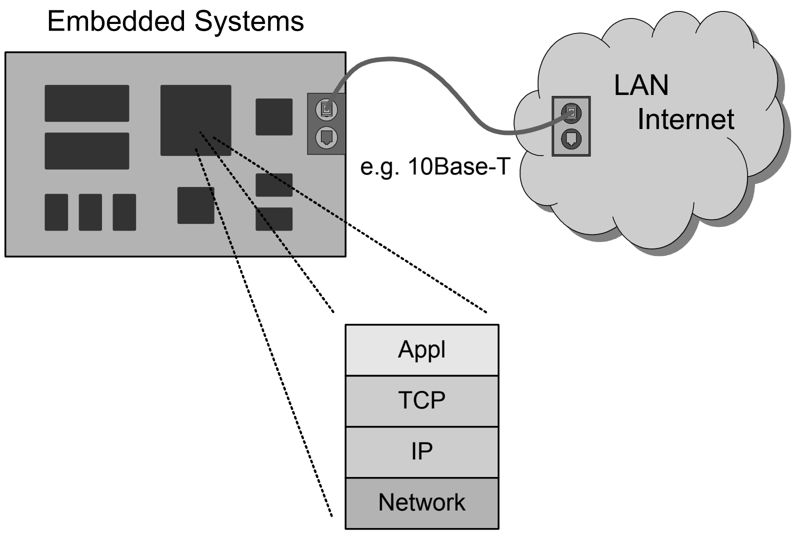

Fig 3: Direct network / internet connectivity via LAN interface

5 Open software architecture for embedded networking applications

In our case study we use an embedded system equipped with *Infineon's C167* low cost microcontroller together with the *Cirrus Logic CS8900* Ethernet controller chip (see fig. 4) in conjunction with the µC/OS-II real time operating system kernel [1] as an open design platform for multithreaded applications with network and internet connectivity.

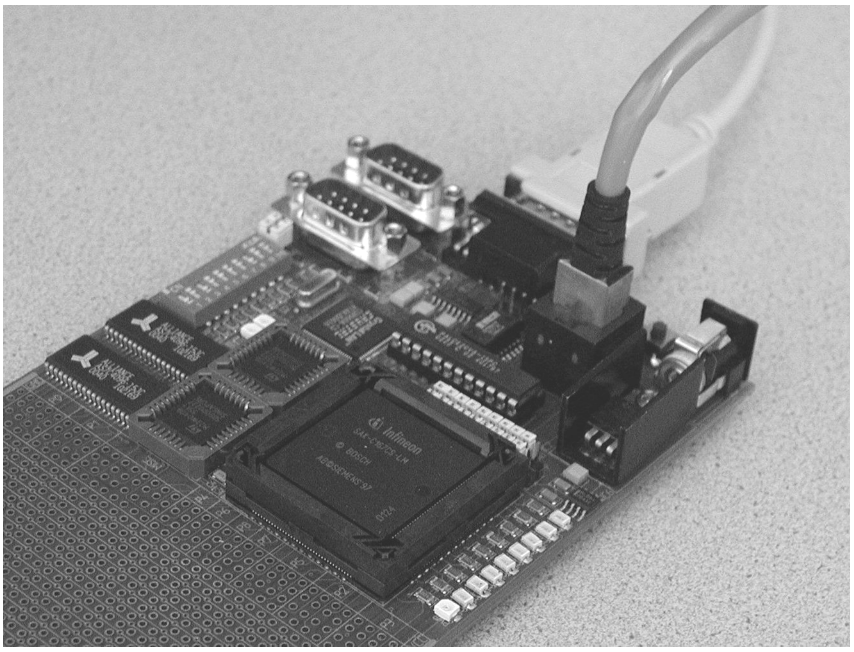

Fig 4: Keil MCBnet-167 evaluation board

The main advantage of this solution in comparison to plug & play solutions like the LANTRONIX XPort and the BECK IPC@CHIP is that the desired network functionality can be customized, even if it is necessary to extend or violate some network protocol standards, which is not unusual in embedded system applications due to specific constraints and limitations.

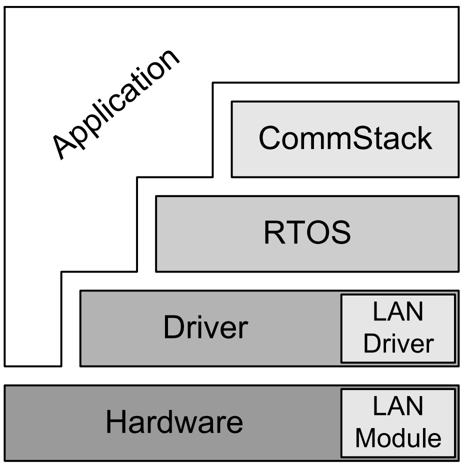

Fig 5: Basic software architecture

From the application's point of view there are several abstraction levels as shown in figure 5. A typical network based application doesn't access directly the LAN-chip hardware or the functions at driver level. Usually a communication stack (e.g. TCP/IP, UDP/IP) provides an appropriate application programming interface (API) to the network tasks of the application. Furthermore, both the application and the communication stack make use of the features of the underlying multitasking- or real time operating system.

5.1 RTOS-based software architecture

The goal of our approach is a transparent integration of the network functionality by means of inter-process communication (IPC) objects offered by almost all multitasking- or real time kernels used in the embedded systems domain. Figure 6 shows the software architecture in detail. The basic hardware abstraction is done by a set of driver functions located in layer A. Layer B realizes an event oriented interface at network frame level to the application (with or without a dedicated communication stack) in layer C. Both layers B and C need operating system (OS) support.

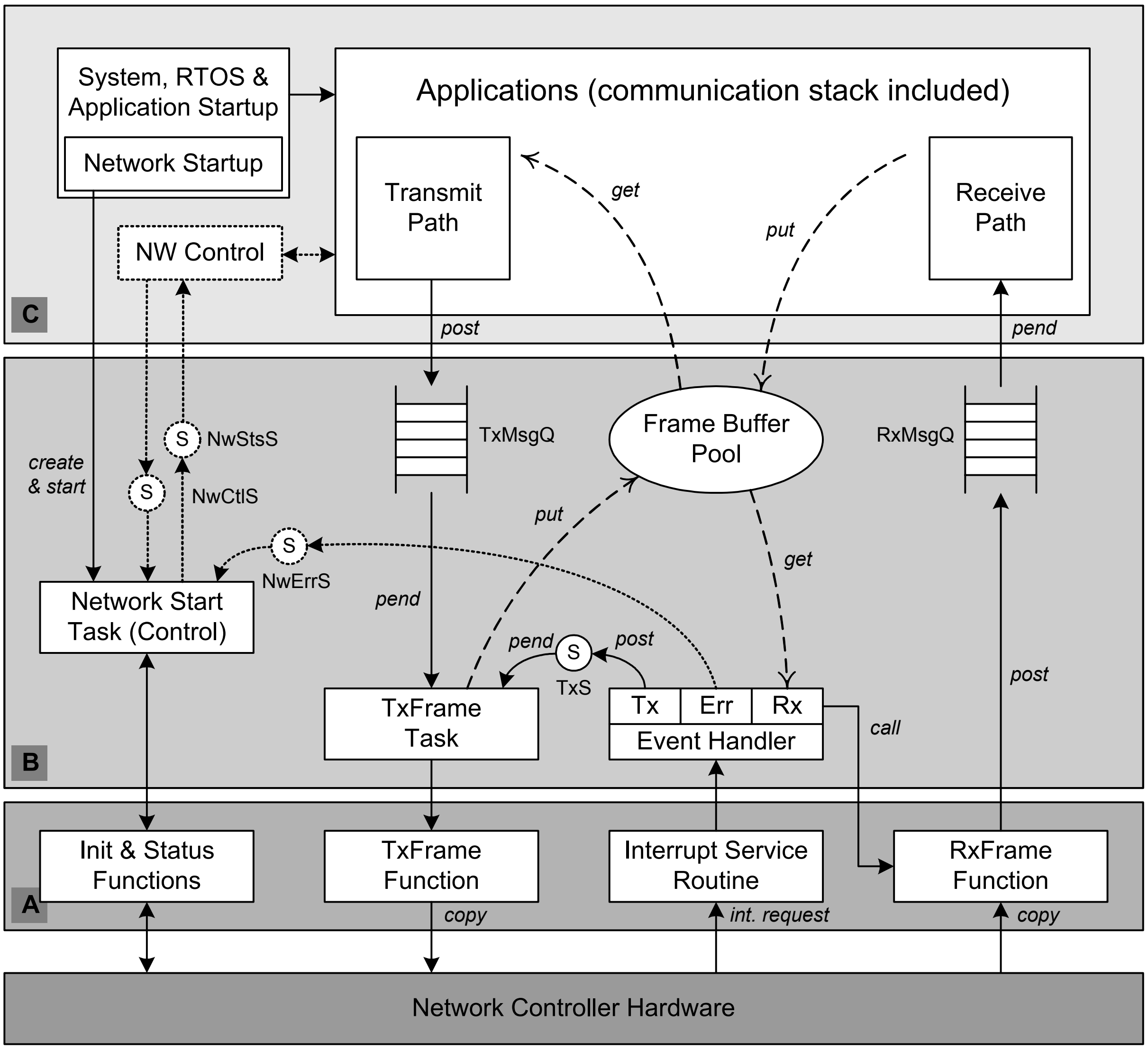

Fig 6: RTOS-based networking software architecture

5.2 The communication paths

After a successful startup of the networking hardware and the objects of layer B a frame based communication can take place. Let's first consider the receive path. After the LAN-controller has received a complete MAC frame the receiver part (*Rx*) of the *Event Handler* is invoked by the *Interrupt Service Routine* which is triggered by a hardware interrupt. A buffer is allocated from the *Frame Buffer Pool* and the *RxFrame* function is called to copy the whole frame from the LAN-controller to the buffer. After that, a pointer to this buffer is posted to the received frame message queue (*RxMsgQ*). This signaling of an event makes a waiting task at layer C ready to run. Depending on the communication stack the frame-payload is processed and delivered to the associated application task. The uppermost level of the communication stack – or maybe the application task itself – is responsible for returning the buffer to the buffer pool when the frame isn't needed anymore.

The transmit path works very similar; an application task or the top layer of the used communication stack allocates a buffer and put in the payload data at an adequate position. The buffer is passed down the communication stack and specific header information is put into the buffer at each level. Finally the buffer's address is posted to the transmit frame message queue (*TxMsgQ*). The *TxFrameTask* is scheduled and the frame is copied to the LAN-controller by means of the *TxFrameFunction*. After that, the buffer is put back to the buffer pool. The transmit semaphore (*TxS*) is necessary to synchronize the *TxFrameTask* to the LAN-controller in order to lock the controller's frame memory unless the previously copied frame is transmitted over the network.

5.3 Buffer management and the zero-copy approach

The *Frame Buffer Pool* is a memory partition with a sufficient number of fix-sized memory blocks large enough to hold the maximum size network MAC frame together with the management header MH (*Extended Frame*, see figure 7a). The management header is not standardized in any way and holds additional information needed for the levels of the communication stack (see figure 7b). Usually the MH holds a set of pointers to the corresponding protocol headers and the start of the payload.

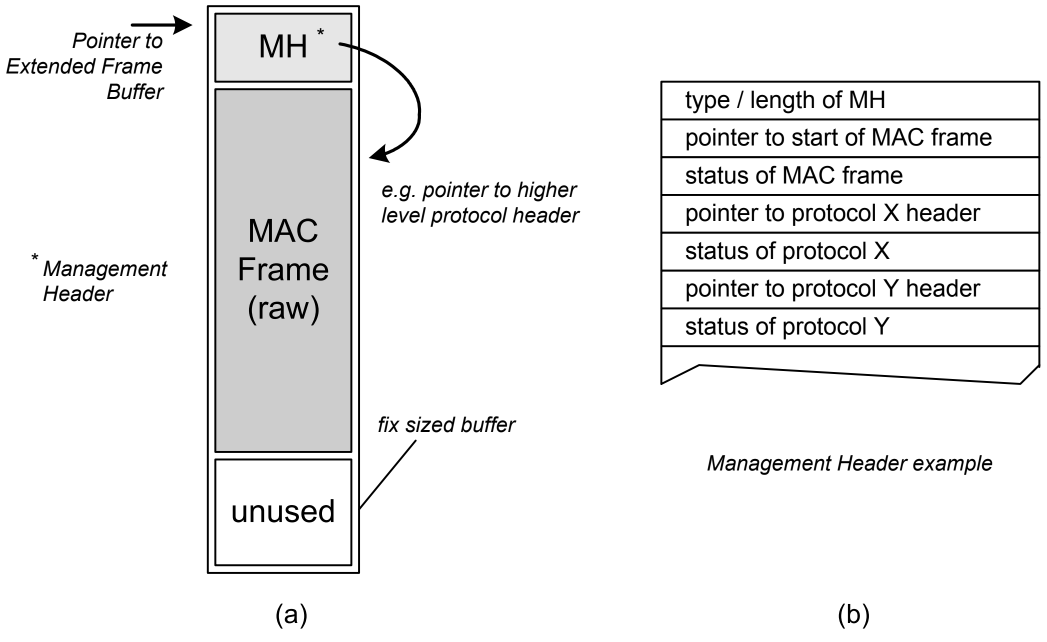
Pointer to Extended Frame Buffer
MH *
MAC Frame (raw)
unused
e.g. pointer to higher level protocol header
* Management Header
fix sized buffer
type / length of MH
pointer to start of MAC frame
status of MAC frame
pointer to protocol X header
status of protocol X
pointer to protocol Y header
status of protocol Y
Management Header example
(a)
(b)

Fig 7: Extended Frame

All the way up and down the levels of the communication stack there is no need to copy any data due to the zero-copy approach. Information is passed from one level to another only by handing over the pointer to the *Extended Frame Buffer*. A certain level knows the position of its relevant fields in the frame by means of the pointers supported in the management header. Each level, of course, has to know the structure of the management header, as it knows the organization of its protocol header as well.

5.4 Initializing, control, status and error handling

To initialize the network the application has to create and start the *Network Start Task*. This is usually done during normal system startup, right after starting the operating system. The *Network Start Task* itself creates all necessary tasks and inter-process communication (IPC) objects and calls the driver level initialization functions (layer A). After successful completion of the *Network Start Task's* main function the event oriented network interface is up and running.

Further control operations and status requests can be handled also by the *Network Start Task* by means of IPC-objects such as semaphores (shown in figure 6) in conjunction with shared memory regions or additional message queues. Layer B might be extended by a dedicated *Network Control Task* to implement a more elaborate network management functionality, e.g. to provide a basis for SNMP.

Networking errors are usually signaled via hardware interrupts by the LAN-controller. The associated *Interrupt Service Routine* (see figure 6) calls the *Err* part of the *Event Handler*, where the network error semaphore *NwErrS* is posted and the *Network Start Task* (or a dedicated *Network Error Task*, not shown in figure 6) is scheduled in order to perform an escalation procedure to inform the afflicted application task.

5.5 OS requirements

As mentioned earlier we use the real time kernel µC/OS-II; currently in the release V2.51 ported for C167 microcontrollers and the Keil-IDE. For implementing the layer B functionality there are some requirements to operating systems in addition to normal task-management:

- ♦ semaphores
- ♦ message queues
- ♦ simple memory management

Basically almost all operating systems fulfill these requirements. If memory management isn't part of the OS, this feature can be realized very easily using the standard functions *malloc()* and *free()*. If the OS doesn't provide message queues, this feature can be realized using circular buffers in conjunction with general semaphores to synchronize the conditions empty and full. If the OS doesn't support semaphores (or semaphore like synchronization objects) it would be a good idea to think about choosing another OS.

6 Conclusion

The open software architecture introduced in this paper is suitable for almost all microcontroller platforms equipped with a LAN circuitry and a ported RTOS to implement a software layer with basic functionality necessary for both communication stacks (e.g. a TCP/IP stack) and flat networking applications.

References

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

[2] Jan Axelson, *Embedded Ethernet and Internet Complete*, Lakeview Research

[3] W. Richard Stevens, *UNIX Network Programming - Volume 1, Networking APIs: Sockets and XTI)*, Prentice Hall