A Lightweight Application Framework for Web Enabled Embedded Systems

AHMED I.SHARAF
Computer Science Dept., Institute of Scientific Research and Revival of Islamic Heritage, Umm Al-Qura University, KSA.

AHMED E. HASSAN
Electrical Engineering Dept. Faculty Of Engineering
Mansoura University. Egypt

ABSTRACT

The spread of embedded systems is growing in very rapidly. Embedded systems are usually suffering from limited resources in terms of processing power, power consumption, memory and storage. These limitations represent a challenge for embedded system developers.
In this paper, a lightweight application framework for embedded systems is presented. The application framework provides the developer with powerful components instead of building them from scratch. Using the proposed framework should increase the productivity and flexibility. It should also minimize the effort and time. The proposed framework includes power management, memory management, event driven mechanism, timer management and TCP/IP compact stack.

Keywords

Embedded systems, Application Framework, Software engineering, Web enabled embedded systems, SysML.

1. INTRODUCTION

Embedded systems are computing systems with tightly coupled hardware and software integration that are designed to perform a dedicated function[1]. An embedded system includes software, hardware and perhaps additional parts either mechanical or electronic designed to perform a dedicated function. Embedded systems are nearly everywhere and varies in size, scope and functionality. Embedded systems could be used in wireless sensor network, DSP, Multimedia, Robotics, medical devices …etc. For example, wireless sensor network is equipped with 8 bit microcontroller, code memory about 100 kilobytes, and less than 20 kilobytes of RAM. These devices could be smaller and cheaper in the future.
Embedded systems are resource limited devices in terms of power, memory, processing power and storage. Networking is a traditional method to get benefits from the limited resources. A networked embedded system is a collection of spatially and functionally distributed embedded nodes, which are interconnected by means of wired or wireless communication infrastructure and communication protocol[2]. Embedded systems can use different methods for networking. At low speeds and short distances within the same electrical enclosure, or when communicating between two or more systems in close proximity, simple communications wiring using parallel or serial data is normally used. These systems are normally described as bus systems, rather than network systems. Conceptually however, they serve a similar purpose. Many microprocessors contain built-in peripherals for inter-device communications

such as serial hardware interfaces: universal asynchronous receive transmit (UART) or inter-integrated circuit (I2C) and others that provide simple one or two-way communications at various speeds. Various other simpler proprietary formats are also available: serial peripheral interface (SPI) or the various single wire interfaces offered by various IC device manufacturers. One other approach is based on global standard such TCP/IP, Bluetooth, Wi-Fi and RFID. Networked embedded systems can be found in many applications such as Industrial Ethernet, Environmental monitoring, Computer peripherals, Alarm and security systems, Voice over IP and much more[3].
Authors of this work represent a general purpose application framework for embedded systems. The application framework provides powerful components for embedded system developers and also provides a modern software engineering model.

1.1. TCP/IP

The TCP/IP protocol suite has become the standard for computer communications in today's networked world, mainly because of its simplicity and power[4]. TCP/IP stands for: The Transmission Control Protocol (TCP)/Internet Protocol (IP), which are the names of the two most important protocols in this suite. The protocol suite was constructed to enable communication between hosts on different networks. Therefore one important aspect is the creation of an abstraction for the communication mechanisms provided by each type of network. TCP/IP hides the architecture of the physical network from the developer of the network application. TCP/IP is modeled in layers as many other networking software. This architecture has many advantages such as ease of implementation and testing, ability to alternative layer implementations etc. Each layer communicates with those above and below through a well-defined interface, and each layer has its own well defined tasks. TCP/IP contains application, Transport network and network interface layers.

The application layer is provided by the program that uses TCP/IP for communication. Examples of applications are Telnet, the File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), Hypertext Transfer protocol (HTTP) …etc. The transport layer is responsible for transferring data from one application to its remote peer. There is support for multiple applications to use the layer simultaneously. The most used transport protocol is Transmission Control Protocol (TCP) which provides reliable data delivery. Another commonly used protocol is User Datagram Protocol (UDP) which provides unreliable data delivery. This layer works as an abstraction of the physical network architecture below it. Internet Protocol (IP) is the most important protocol in this layer. IP does not provide reliability or error recovery; this is up to higher layers to handle. Other network layer protocols are ICMP, IGMP, ARP and RARP. The Network interface layer works as the interface to the actual network hardware. TCP/IP does not specify any protocol here, but can use almost any network interface available, which illustrates the flexibility of the Network layer.

1.2. MULTITHREADING

Most of computer systems can perform several tasks at the same time. While running a user program a computer can also be reading from disk or printing text to output device. In multiprogramming system, the CPU performs switching from program to another. Each program can executed for a time slot of tens to hundreds of milliseconds. In multiprogramming model all the running software on the computer are organized into a number of sequential processes. Many modern operating systems now provide features enabling a process to contain multiple thread of control. Thread is a basic unit of CPU utilization, it comprise a thread Id, program counter, register status and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals[5]. The benefits of multithreaded programming can be broken down into four major categories:

- Responsiveness
- Resource sharing
- Economy
- Utilization of multiprocessor architectures

Although it seems that multithreading is very useful, thread have the following drawbacks[6]:

- Writing multithreaded programs requires very careful design. The potential for introducing subtle timing faults, or faults caused by the unintentional sharing of variables in a multithreaded program is considerable.
- Debugging a multithreaded program is much harder than debugging a single-threaded one, because the interactions between the threads are very hard to control.
- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine, as there are only so many CPU cycles to be had, though if nothing else is trying to execute and you have multiple processors or a hyper-threaded CPU, a multiple thread approach may offer benefits.
- Multithreading requires stack per thread implementation which could be a memory overhead for memory constrained systems such as embedded systems. For this reason, many operating systems for sensor networks, including TinyOS[7, 8], SOS [9], and ConTiki [10]are based on an event-driven model.

It's clear that multithreading is not an easy task to do for embedded systems. Another method to perform concurrency is through event driven programming.

1.3. Event driven mechanism

Event based programs[11]are organized around the processing of events. When a program cannot complete an operation immediately because it has to wait for an *event* (e.g., the arrival of a packet or the completion of a data transfer), it registers a *callback* (a function that will be invoked when the event occurs). Event based programs are typically driven by a loop that polls for events and executes the appropriate callback when the event occurs. A callback executes indivisibly until it hits a blocking operation, at which point it registers a new callback and then returns.

Event driven programs require a series of small callback functions, one for each blocking operation. Any stack allocated variables disappear across callbacks. Thus, event driven programs rely heavily on dynamic memory allocation[8] and are more prone to memory errors in low level languages such as C and C++. As an example, consider the following asynchronous write function:

```
void awrite (int fd , char *buf , size_t  size , void (*cb)
(void *), void *arg);
```

awrite might return after arranging for the following to happen: as soon as the file descriptor *fd* becomes writeable, write the contents of *buf* to the descriptor, and then call *cb* with *arg*. *arg* is state to preserve across the callback state that likely would be stack allocated in a threaded program.

1.4. Portability

As the number of embedded systems platform architecture increases, it is desirable to have a common software infrastructure that is portable across different platforms. The proposed framework does not depend on specific architecture or specific platforms. Although some a few components such device driver is not portable enough, the other components are. The proposed framework is based on ANSI C language which is supported by most embedded manufacture. The single unifying characteristics of today's platform are the CPU architecture which uses a memory model without segmentation or memory protection mechanism. Program code is stored in reprogrammable ROM and data in RAM. The proposed framework is designed so that the only abstract provided by the base system is CPU multiplexing.

2. RELATED WORK

Authors and software vendors proposed various number of software components for embedded systems. These components range from simple application to real time operating system. Lightweight TCP/IP stack is very important stack which is developed by many authors and software vendors.

Atmel provided an AVR internet toolkit[12], which can be used for 8 bit embedded internet applications. This toolkit is suitable for the manufacture products only. Aittamaa et al proposed a modular TCP/IP stack for embedded systems with a tiny timber interface[13]. Simon work added Point to Point Protocol (PPP) for embedded TCP/IP stack. Also Jakobsson et al developed a TCP/IP stack for a real time embedded systems[14].

Adams Dunkels presented two proposals for this problem which are called UIP and IwIP[15]. His proposals are based on redesigning TCP/IP as lightweight separate software that is targeting tiny 8 bit microcontrollers. There are three weak points in Dunkels's proposals. First, there is no software model. Since communication protocols are complex software, modeling is necessary process. Software modeling and analysis can improve system maintenance, flexibility, extensibility and ease of system understanding. Second, both UIP and IwIP are targeting tiny embedded systems that make it difficult to use the solution for anther microcontroller architecture. Lastly, there is no modularity in which makes it hard to customize the functionality of system.

In the following sections, authors present system overview in section 3. Section 4 presents the kernel structure. Section 5

presents the service layer. Section 6 presents the results. Section 7 presents the conclusion and future work.

3. SYSTEM OVERVIEW

This research paper presents a general purpose lightweight application framework for embedded systems. The proposed framework is grouped into three main layers. Each layer includes internal components as shown in Fig1.

The kernel layer handles the lowest level of software. The components of this layer are authorized to access the hardware directly. Each component should handle a subset of problems. The kernel layer includes device driver, power management, memory management, event driven mechanism and timer management.
The service layer contains a networking stack component. This component contains a lightweight TCP/IP stack. The TCP/IP stack is used as an infra-structure communication protocol which could help in networking and accessing the Internet.
The application layer contains the end user application. This application represents an interface between the end user and the embedded system itself. Section 4 discusses the components of the kernel layer. Section 5 discusses the components of the service layer.

3.1. Kernel structure

In this section, the components of the kernel are discussed. The components are discussed as follows:

- Device driver

Most embedded hardware requires some type of software initialization and management. The software that directly interfaces with and controls this hardware is the device driver. The proposed framework includes a default Ethernet device driver which could be used to access the Internet. The default Ethernet driver includes basic 4 categories. Direct hardware accessing category which performs general operations i.e.: chip reset, read from register and write to register. The Initializing and status category which is used to, initialize the driver and report the status of last operation. The Data transmission category which is used to read from DMA registers or writes to DMA registers. Data Transmission is also responsible for polling the drivers and handle interrupts.

- Power management

Energy is becoming a critical resource to not only small battery-powered devices, but also to large systems. High energy consumption is translated to heat dissipation, which in turns increases the cooling cost and could cause the system to failure[16]. The proposed framework assumes 4 states for power modes which are idle, power-down, power-save and standby modes as shown in Fig 2.

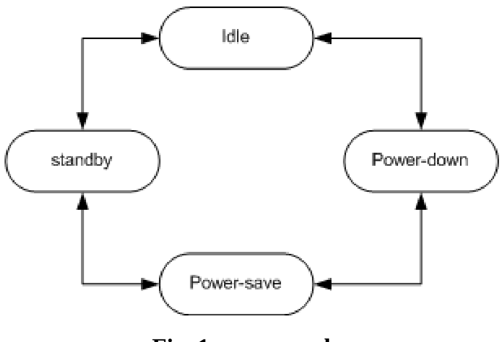

Fig. 1 power modes

The idle power mode is used to stop the MCU only. The other components such as serial ports, watchdog timer, timers could work if they are enabled. If interrupts occurs the MCU will wake up.

In the power-down mode, the external oscillator is stopped while the External Interrupts, the Two-wire Serial Interface

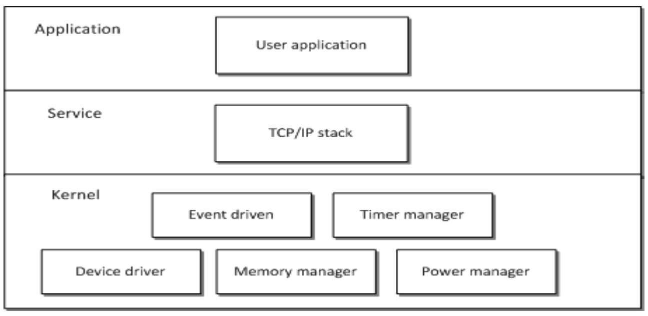

Fig 2 Proposed framework architecture

address watch, and the Watchdog continue operating (if enabled). Only an External Reset, a Watchdog Reset, a Brown-out Reset, a Two-wire Serial Interface address match interrupt, an External Level Interrupt, or an External Interrupt can wake up the MCU. This sleep mode basically halts all generated clocks, allowing operation of asynchronous modules only.
The power-save mode is similar to power down mode, but with one exception. The power save mode enables the timers overflow. The device can wake up from either Timer Overflow or Output Compare event. The standby mode is also similar to power down mode, but with one exception. In this mode the oscillator keeps running.

- Memory management

Memory is the most important resource in embedded system. Embedded systems usually suffer from limited memory. The used approach is to use one single buffer for holding packets. The buffer includes blocks which are handled by the memory manger. When a packet is arrived the device drive places it in a global buffer (Ethernet controller buffer). Then the device driver sends a notification message (interrupt) to the kernel. If the packet is verified and contains data the kernel notify the corresponding application to make one option from the following:

1. Perform online processing on global buffer.
2. Copy the packet contents to secondary buffer and perform the processing on it

The memory manager is based on custom allocation/de-allocation of memory blocks instead of dynamic memory allocation. The memory manager includes **memb_init**, **memb_alloc** and **memb_free**. The **memb_init** accepts one argument of memory block data type and is used to declare that memory block. The **memb_alloc** is used to allocate memory block into the memory. The **memb_free** is used to de-allocate n bytes from the block.

- Event driven mechanism

The proposed framework is based on Protothread technique[9, 17, 18]. Protothread is a new abstraction of programming which could be used in embedded systems. Protothread makes it possible to write event driven programs in a thread like style, with a memory overhead of only two bytes per Protothread.
Protothread can reduce the complexity of a number of widely used programs previously written with event-driven state machines. For the examined programs the majority of the state machines could be entirely removed. In the other cases the number of states and transitions was drastically decreased.

With Protothreads the number of lines of code was reduced by one third. The execution time overhead of Protothread is on the order of a few processor cycles. The proposed model provides conditional block wait statement using **PT_WAIT_UNTIL**, which is intended to simplify event driven programming for memory, constrained embedded systems. The operation takes a conditional statement and blocks the Protothread until the statement evaluates to true. If the conditional statement is true the first time the Protothread reaches the **PT_WAIT_UNTIL** the Protothread continues to execute without interruption. The **PT_WAIT_UNTIL** condition is evaluated each time the Protothread is invoked. The **PT_WAIT_UNTIL** condition can be any conditional statement, including complex Boolean expressions. For example, Fig 3 shows how to use Protothread mechanism for certain scenarios.

a)
PT_BEGIN
//user code
PT_WAIT_UNTI
L(cond1)
// user code
PT_END
	b)
	PT_BEGIN
	//user code
	while (cond1)
	PT_WAIT_UNTI
	L(cond1 or
	cond2)
	//user code
	PT_END
		c)
		PT_BEGIN
		//user code
		if (condition)
		PT_WAIT_UNTIL(
		cond2a)
		else
		PT_WAIT_UNTIL(
		cond2b)
		//user code
		PT_END

Fig 3 pseudo-code of Protothread sequence, iteration and selection operations

- Timer management

  Time is an important aspect in any communication process. Timer management is used for various reasons in communication protocol such as calculating time out operation, retransmission conditions, and packet loose. The proposed framework considers two views of timers high level and low level timers. The high level timer represents the time from the software point of view. The high level timer defines the **clock_time_t** as base data type and defines **CLOCK_SECOND** as terms of time. The timer data-type defines start and interval attributes as **clock_time_t**. The high timer also contains basic functions of timer management such as **timer_set**, **timer_reset**, **timer_restart**, and **timer_expire**. Fig 4 shows how to set a timer for 2 seconds.

```
struct timer periodic_timer;
timer_set(&periodic_timer, CLOCK_SECOND * 2);
if(timer_expire(&periodic_timer))
//user code after timer expires
```

Fig4 pseudo-code of timer manager example

3.2. Service

The service layer includes TCP/IP stack. The TCP/IP stack is very important for connecting the Internet. Although TCP/IP includes various number of communication protocols, the core protocols are implemented. The implemented protocols of TCP/IP stack are ARP, IP, ICMP, TCP and UDP. The service layer keeps polling the device driver until an Ethernet frame is received. The flow chart of this sequence is shown in Fig 5. The *mac_Init* is used to initialize the ARP module and set default values used in the module. The *mac_handler* is used to handle the received ARP frame. Check the data validation and integrity. Extract the data payload and do necessary processing. The *mac_ReqARP* is used to broadcast an ARP request. This function generates the broadcast frame and waits until time out or replay is sent. The *mac_FillHdr* is used to encapsulate Ethernet frame with MAC header and perform physical transmit.

IPH_Init is used to initialize the IP packet and set the standard values. *IPH_Handler* is used to handle the received IP packet and perform necessary processing. Extract the payload from the IP packet and notify the corresponding application. *IPH_FillPacket* is used to fill the packet with payload. *IPH_FillHdr* is used to fill the packet header with both payload and static values. *IPH_CheclSum* is used to perform validation on the packet and calculate the result checksum. The user starts by calling *IPH_Init* to define the packet and then call *IPH_FillPacket* to pass the payload from the higher application to that packet. *IPH_FillHdr* is called to encapsulate the payload and static values of the packet and format the IP header. *IPH_CheckSum* is called to check the validation of IP header. If the value of the checksum fails, the packet is ignored. If the IP header is correct, the packet is passed to device driver to be transmitted physically.

ICMP is a diagnostics protocol which is based on IP packets. ICMP is used to test basic connectivity between hosts. The ICMP should at least be able to respond to the PING request and have enough storage to store data block. The service layer should automatically send response to sender host without waiting for user input. ICMP is handled by the *IPH_handler* which is responsible for handling IP packet.

TCP_Init is used to initialize the TCP packet and set standard values of the packet. *TCP_Handler* is used to handle the received packet. Check the packet validation and extract the payload. *TCP_User* is the user application. *TCP_TX* is used to send data segment to destination. This method doesn't send the segment physically, it just encapsulate both data and header and pass it to DMA. *TCP_Timer* is used to set the timer value and detect connection timeout. *TCP_Putch* is used to send one character to DMA. Fig 6 shows TCP sequence diagram. This sequence diagram includes 6 lifelines and 3 optional scenarios. First scenario represents how the system sends TCP segments.

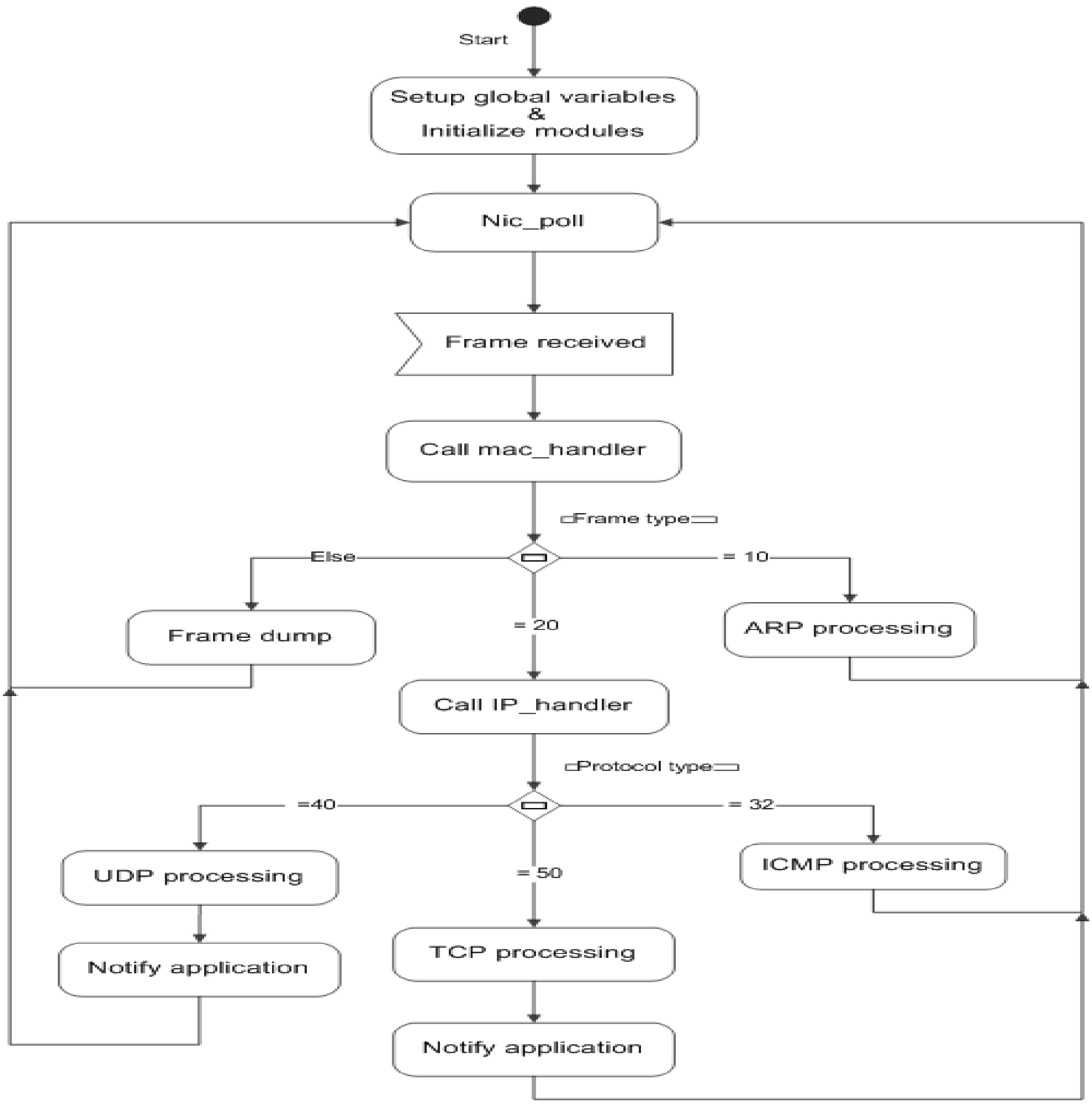

Fig 5 The TCP/IP program

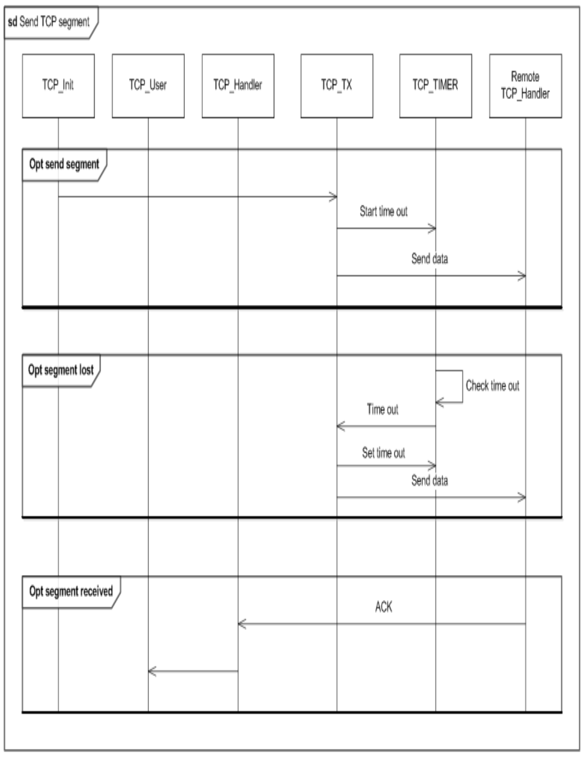

Fig 6 TCP sequence diagram

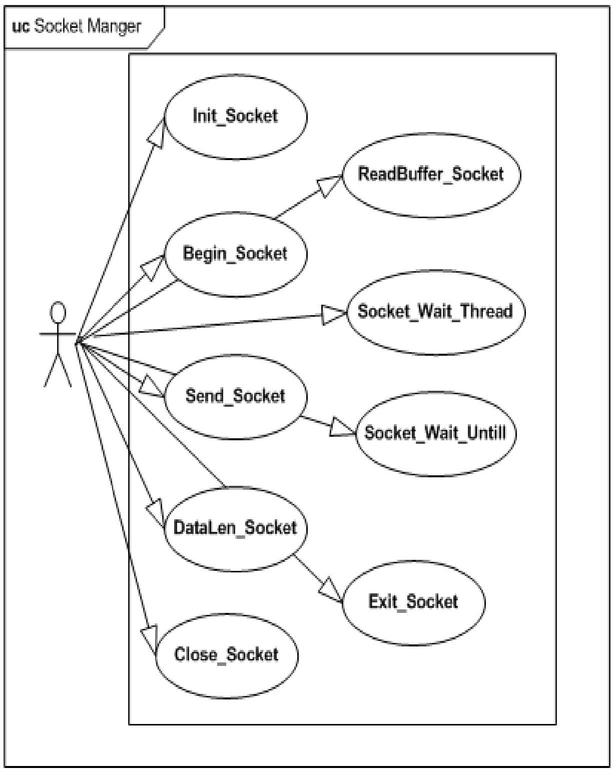

Fig 7 use case of socket component

Table 1 Static analysis of the proposed framework

	Statements	Average Statements Per Functions	Max Cyclomatic complexity	Max depth	Average depth	Average complexity
Time manager	28	2.5	1	1	0.25	1
Event driven manager	32	0	0	1	0.03	0
Memory manager	35	10.3	4	4	1.03	2.67
TCP/IP stack	455	31.9	9	5	1.42	2.75
Ethernet driver	194	28.5	9	3	1.07	3
Total	**744**	**73.2**	**23**	**14**	**3.8**	**9.42**

Send Segment start with *TCP_Init* method to declare the packet and set the default values. Then *TCP_TX* is called to send the packet to the DMA, when the packet is sent the timer start counting until timeout condition.

Second scenario represents how the system handles loss of segment. *TCP_Timer* keeping ticking until timeout condition, when timeout occurs without remote host replay the *TCP_TX* resend the segment again and reset the timer.

Last scenario represents the system behavior when TCP segment is received. *TCP_Handler* is called to handle the received segment, and then *TCP_User* is notified. UDP is a simple transport communication protocol. UDP is nothing more than IP datagram with few added fields. UDP can be a very effective protocol for the transmission of simple half-duplex style data. Each operation by the application process produces one data block, which is converted to one UDP datagram, which causes one IP datagram to be transmitted.

UDP is a simple transport communication protocol. UDP is nothing more than IP datagram with few added fields. UDP can be a very effective protocol for the transmission of simple half-duplex style data. Each operation by the application process produces one data block, which is converted to one UDP datagram, which causes one IP datagram to be transmitted.

The *UDP_Init* is used to initialize the UDP module and usually called only once at the program start. The *UDP_handler* is used handle the received UDP packet and extract the payload. The *UDP_Tx* is used to generate and transmit UDP datagram. The *UDP_User* is the user application.
Socket component is an end point of bidirectional communication between hosts or between processes. The use case of socket component is shown in Fig 7. Socket component includes *Init_Socket* which is used to declare the socket and determine its buffer. *Begin_Socket* is used to start the thread associated with certain socket. *Send_Socket* is used to send the socket data. The socket thread blocks the socket access until the data has been sent. *DataLen_Socket* is used to get the previously read data. *Close_Socket* is used to terminate the communication between hosts. *ReadBuffer_Socket* is used to read the buffer declared within the socket. *Socket_Wait_Thread* is used to wait specific thread to execute specific condition. *Socket_wait_Untill* is used to wait for specific condition.

4. RESULTS

The development of embedded systems constrains the compact of code size in order to give user application much room. The discussed metrics shows how the proposed framework is lightweight. Table 1 shows the static metrics of each component in the proposed framework. The device drive component is amachine dependent one and should be changed when hardware change.

5. CONCLUSION AND FUTURE WORK

Authors of this work presented a general purpose framework for embedded systems. The proposed framework should help embedded systems developers and software engineers to build powerful and effect applications. The proposed framework consists of 3 layers model. Each layer includes essential components for embedded systems developers. The proposed framework contains power management, memory management, event driven mechanism and lightweight TCP/IP stack. The proposed framework is based on modules which make it easy in maintenance and extensible.
Authors plan to extend the framework components and enhance the power management module. The framework could be enhanced with many components such as security and cloud support.

6. REFERENCES

[1] Q. Li, C. Yao, Real-Time Concepts for Embedded Systems, CMP Books, 2003.

[2] R. Zurawski, Embedded Systems Handbook, 2009.

[3] F. Eady, Networking and Internetworking Wirh Microcontrollers, Elsevier/Newnes, 2004.

[4] P. Loshin, TCP/IP Clearly Explained, Elsevier Science, 2003.

[5] A. Silberschatz, P.B. Galvin, G. Gagne, Operating System Concepts with Java, John Wiley & Sons, Incorporated, 2011.

[6] N. Matthew., R. Stones, A. Cox, Beginning Linux Programming, 3rd Edition, 2003.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, System architecture directions for networked sensors, SIGARCH Comput. Archit. News, 28 (2000) 93-104.

[8] P. Levis, D. Gay, TinyOS Programming, Cambridge University Press, 2009.

[9] C.-C. Han, R. Kumar, R. Shea, E. Kohler, M. Srivastava, A dynamic operating system for sensor nodes, in: Proceedings of the 3rd international conference on Mobile systems, applications, and services, ACM, Seattle, Washington, 2005, pp. 163-176.

[10] A. Dunkels, B. Grönvall, T. Voigt, Contiki - a lightweight and flexible operating system for tiny networked sensors, in: The Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I), Tampa, Florida, USA, 2004.

[11] T. Faison, Event-Based Programming, Apress, 2006.

[12] Atmel, www.atmel.com, in, 2011.

[13] A. Simon, R. Isak, A modular TCP/IP stack for embedded systems with a tinyTimber interface, in, Lulea University of Technology,Sweden., 2007.

[14] S. Jakobsson, E. Dahlberg, Development of a TCP/IP Stack in real time embedded system, in, Umea University, Department of Computing Science, Sweden., 2007.

[15] A. Dunkels, Full TCP/IP for 8-bit architectures, in: Proceedings of the 1st international conference on Mobile systems, applications and services, ACM, San Francisco, California, 2003, pp. 85-98.

[16] H. Huang, K.G. Shin, C. Lefurgy, K. Rajamani, T. Keller, E. Hensbergen, F. Rawson, Software–Hardware Cooperative Power Management for Main Memory, in: B. Falsafi, T.N. VijayKumar (Eds.) Power-Aware Computer Systems, Springer Berlin Heidelberg, 2005, pp. 61-77.

[17] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, J.R. Douceur, Cooperative Task Management without Manual Stack Management or, Event-driven Programming is Not the Opposite of Threaded Programming, in: In Proceedings of the 2002 Usenix ATC, 2002.

[18] A. Dunkels, O. Schmidt, T. Voigt, Using Protothreads for Sensor Node Programming, in: In Proceedings of the REALWSN 2005 Workshop on RealWorld Wireless Sensor Networks, 2005.