# BUILDING A POWER EFFICIENT SYSTEM WITH RTOS, TCP/IP STACK AND FILE SYSTEM

Vladica Sark[1)], Josif Kosev[2)], Faculty of electrical engineering and information technology, Karpos bb, 1000 Skopje, Macedonia, email: [1)] sark@feit.ukim.edu.mk; [2)] j.kosev@ieee.org

**Abstract** - *Embedded systems are getting more and more demanding and require more and more energy for the tasks they are intended to do. Usually today's embedded systems are required to have more and more peripherals, which on the other hand requires more and more energy consumption, more memory and more powerful processor to utilize. This paper presents a solution for this kind of embedded system that has real-time operating system, TCP/IP stack, FAT file system and at the same time is poor with memory and processing power, which allows less energy consumption. All this functionalities are implemented on not so powerful ARM7 controller. Also, capabilities for energy saving are considered. This energy saving capabilities are intended to be included in the operating system and to be used without direct interaction of the user code. At last, an operation system is ported for this ARM7 controller and TCP/IP stack and file system are embedded together with it. Also, power saving features is added to this operating system. The conclusion presents how much resources are needed for this system to operate.*

## 1. INTRODUCTION

As we know, today, embedded systems are playing a big role in our life. Everything is "digital" and everything is attempted to be microprocessor controlled. Not being microprocessor controlled, makes one system outdated and not attractive.

Today, on the market of microprocessors and microcontrollers there are a dozen of players, offering a palette of hundreds of different microprocessors and microcontrollers. Starting from the simplest eight bit microprocessors up to multicore processors for embedded applications, we have a lot of choices for choosing the right processor for our application. Usually, the powerful microprocessors like OMAP, ARM9 and so on, come with Linux, Android, Windows CE support. Of course, there are more operating systems, for these microprocessors, than we can imagine, but the application, in which the microprocessor is going to be used, limits the choice on a fewer operating systems. Even, today, we have very powerful microprocessors, which are usually widely used in mobile phones, video cameras, photo cameras, set top boxes and so on, the demand for the low end microprocessor, embedded in microcontrollers, is still quite large. This actually can be seen from the manufacturer's palette of offered products. Big part of their business is actually comprised from the low end microcontrollers. There is more than one reason for this. Actually, there are a lot of reasons, but we will discuss only a few, which we think that are maybe most important. The users that use these microcontrollers usually want to build a system, which is going to be as simple as possible, but still enough powerful for the task intended for. This will make the development less complex and at the same time less expensive. Size also plays a big role in the whole concept. Usually, here it does not mean that good things come in small packages. Powerful microprocessors, in most cases, need an external RAM and flash memory, so they are required to have address bus and also data bus. This increases pin count and also, it is a reason for more complicated PCB. Finally, the last, but not the least significant reason for using low end microcontrollers, whenever possible, is the price of the chip.

Today, there are a bunch of cheap microcontrollers, with pretty, good performances. One of them are ARM7 based microcontrollers from different manufacturers. One example is ARM7 form NXP. These microcontrollers comes with a lot of integrates peripherals, also including limited RAM and flash memory, for very reasonable price. Of course, putting some modern embedded operating system, like Linux, on these microcontrollers is impossible, only because of limited RAM and flash memory. This can be a problem, if a complicated application is intended to run on this kind of system. Using operating system, make things simpler, because the developers of the software for the system can only write separate tasks, without needing to learn the microcontroller. Of course, if there are strict timing requirements, it is necessary to have a real-time operating system.

This paper describes, how one embedded real-time operating system was ported for a specific processor and also how that operating system was upgraded with TCP/IP stack, file system and some power saving features.

The overall system was actually built on one chip with internal RAM and flash memory, what makes that system very useful for cheap applications.

## 2. PORTING RTOS TO AN ARM7 SYSTEM

The RTOS (Real-time operating system) that we have chosen to port on a ARM7 (LPC2378 exactly) system is FreeRTOS. There are more reasons why this RTOS was chosen to be used. In table 1 there are some characteristics that this RTOS have and some other RTOSs, even commercial ones, does not have.

Table 1 – FreeRTOS functionalities

| No. | RTOS functionality |
|---|---|
| 1 | Free RTOS kernel - preemptive, cooperative and hybrid configuration options. |
| 2 | Official support for 23 architectures (counting ARM7 and ARM Cortex M3 as one architecture each). |
| 3 | FreeRTOS-MPU supports the Cortex M3 Memory Protection Unit (MPU). |
| 4 | Designed to be *small, simple and easy to use*. Typically a kernel binary image will be in the region of 4K to 9K bytes. |
| 5 | Very portable code structure predominantly written in C. |
| 6 | Supports both tasks and co-routines. |
| 7 | Powerful execution trace functionality. |
| 8 | Stack overflow detection options. |
| 9 | No software restriction on the number of tasks that can be created. |
| 10 | No software restriction on the number of priorities that can be used. |
| 11 | No restrictions imposed on priority assignment - more than one task can be assigned the same priority. |
| 12 | Queues, binary semaphores, counting semaphores, recursive semaphores and mutexes for communication and synchronisation between tasks, or between tasks and interrupts. |
| 13 | Mutexes with priority inheritance. |
| 14 | Free development tools (Cortex-M3, ARM7, MSP430, H8/S, AMD, AVR, x86 and 8051 ports). |
| 15 | Free embedded software source code. |
| 16 | Royalty free. |
| 17 | Cross development from a standard Windows host. |
| 18 | Pre-configured demo applications for selected single board computers allowing 'out of the box' operation and fast learning curve. |

As can be seen from table 1 there are a lot of functionalities and characteristics that FreeRTOS has and what some other RTOSs does not have. For example, the first functionality is very important and allows the user to control how the RTOS is going to threat different situations. FreeRTOS has ability to work as a preemptive, cooperative and as a hybrid RTOS. This is not possible with a lot of RTOSs.

Functionalities 10 and 11 from table 1 are also important. It is very common case when user needs to have 2 tasks with same priority. Some commercial RTOSs are only able to assign one priority per task.

Porting this RTOS on ARM7 architecture was not very hard, because it already exists port for the ARM7 architecture.

Our port was designed to run on an LPC2378 from NXP. This microcontroller has 56 Kbytes of RAM and 512 Kbytes of flash memory. It also have a lot of other peripherals like: DMA, USB, Ethernet, $I^2C$ SPI, GPIO, UART etc. that makes it ideal for this RTOS. It is also cheap and does not require external peripherals to run, what makes possible to build very simple system with capabilities like some modern systems. Of course, the price we pay is the limited speed and limited memory resources. But, there are a lot of applications which does not require so much memory resources and speed, what makes this system ideal for them.

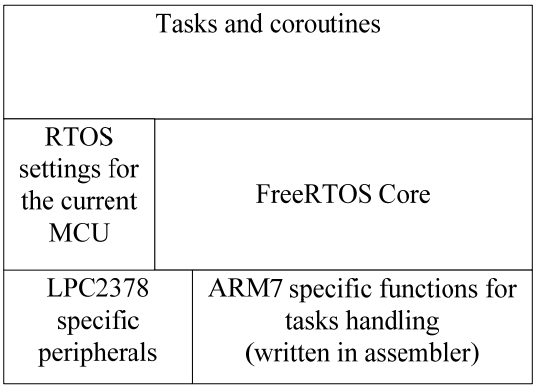


Figure 1 – FreeRTOS structure

On figure 1 is shown how the FreeRTOS code is organized. At the bottom of the figure is the part of the operating system that is architecture specific. There are functions that access the peripherals, in our case timers and interrupt controller, which are used for task switching. The other part in the bottom is for the functions that actually implement task switching. This part is done in assembler, because it is microcontroller specific and also it does some things that C compiler would not allow. In this case, only the part for the peripherals was written from scratch. The part that is specific for the ARM7 microcontrollers was not changed because LPC2378, that is used, is ARM7 based. The FreeRTOS core is written in C. It is intended to be portable, so at this stage it doesn't have to be changed. RTOS settings are something that should be adopted for every microcontroller, even maybe, for every application. It mainly concerns the memory model of the microcontroller. In that part the memory model for each task is specified and also a heap for all tasks is specified. That part of the RTOS was also written when porting the RTOS. Finally, the top level on the figure is user specific at this stage. Users, that use this port, define their own tasks here.

## 3. INTEGRATING LWIP TCP/IP STACK

The lwIP (Light Weight Internet Protocol) stack is an IP stack intended to be used on embedded systems. It is a free and open souce TCP/IP stack and supports all the important functionalities of the TCP/IP v4 protocol. It is being actively developed and new versions are coming over and over. This means that all important new functionalities are being added. As the name suggests, the lwIP is light weight, which means that it is very suitable for using it on some embedded system with limited resources.

The lwIP stack is intended to be used with operating system, even it can be used without it, so the code is organizes as one task that should be started from the operating system. On the figure 2 there is a simple illustration of how the stack is organized. As can be seen, the stack is mainly consisted of three parts. The lowest part is hardware specific. It is used to access the hardware that is responsible for sending and receiving Ethernet packets. The LPC2378 has

a media independent interface (MII) on which is connected an external Ethernet chip. The lowest part on the figure 2 is used for setting up and accessing MII. It is responsible of sending and receiving Ethernet packets in which are encapsulated IP packets.

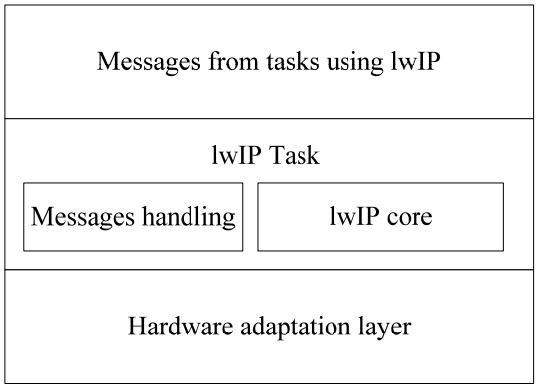


Figure 2 – lwIP structure

In our system a KS8721BL chip is used. This chip is a fully compliant with IEEE 802.3u standard. It supports 100BASE-TX/100BASE-FX/10BASE-T physical layer. Because this part is specific for the used hardware, it has been written especially for the hardware used. It is actually one file with few functions in it, that set up registers for MII, registers on the KS8721BL (the Ethernet chip) chip and send and receive Ethernet packets to and from the MII interface.

If, in the future, other chip is going to be used with this system, only this file should be adjusted.

The middle part from the figure 2 is actually the lwIP. It is consisted of one task that runs the lwIP core and communicates with other tasks requiring lwIP access. This part is used as is and the only thing that is configured is the priority of this task, what proved to be very important.

Finally, the highest part of the figure 2 is the part that is used to send messages. FreeRTOS has implemented "message queues" mechanism for intertask communication. Of course, creating, formatting and sending message via the queue, directly, is not elegant solution. Because of that, lwIP has functions, which accept TCP and UDP packets, from user tasks, and create messages that are passed to the lwIP task. So, the user has feeling that he is putting TCP and UDP packets directly in the lwIP stack, without using "message queues".

It seems at first that this solution, with separate task for lwIP and message queues, is not happily chosen and it is too complicated. From the user perspective, it is not so complicated. The user that sends and receives IP packets does not have an idea what happens in the background. For porting this system, it might be a little confusing at first, but this solution actually enables more than one task to send IP packets simutaniously. If the lwIP access was not solved this way, a mess would happen when more than one task would try to access it simultaneously.

## 4. INTEGRATING FILE SYSTEM

As it was mention earlier, integration of the file system has been done for this system. Of course, there is a wide choice of file systems available, including commercial and open source. For our purpose, EFSL (Embedded File System Library) was chosen. This is open source file system, free, of course and not so popular. It is not popular, because it development is not active for few years and it supports only FAT12, FAT1 and FAT32. It is also ported for a few microcontrollers and does not have support for operating system. Of course, there are some good stuff why this file system was chosen. It is pretty simple, lightweight and does not require much effort to be ported. On figure 3, there is the structure of the final port for LPC2378.

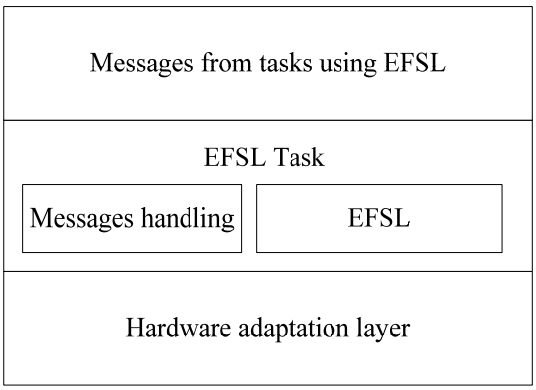


Figure 3 – Final structure of the ported file system

The only thing that has been used as is, is the EFSL library. The other stuff was written from scratch and as can be seen, they have the same structure as the lwIP.

This system is intended to be used with external SD card, because this card is widely used in consumer electronics, is easily affordable and because LPC2378 has SD card interface with dedicated hardware for SD card access.

The hardware adaptation layer has been written and it was used to initialize the SD peripheral, to get the properties of the SD card inserted them in the system and to read and write sectors from the SD card. From here on, these functions are used directly by EFSL and user does not need to use them. The rest of the structure is like the lwIP structure, from the same reasons mentioned there.

## 5. ADDING POWER SAVING FEATURES TO FREERTOS

FreeRTOS, as is, does not include power saving functionality. In today's systems, power efficiency is important feature, because, a lot of systems derive the power from energy limited sources, like batteries.

The first thing that has been done in solving this issue, is putting the processor in sleep mode whenever possible. This is done very easy if there is no operating system, but when RTOS is employed, it is not so simple. The problem is that a task should not put the processor in sleep mode, because there might be some other task, with lower priority, waiting to be executed.

To solve the issue mentioned, FreeRTOS core source was examined. The core, itself, has an idle task, which is called whenever there is no other task requiring service. This task is used to clean memory, if there is task that is disposed (erased). So, after performing the memory cleaning, the processor can go in sleep mode, until next timer tick issues wake up. Using this simple procedure, the processor can go in sleep mode, every time when the idle task is started. Of course, turning off peripherals is more complicated task. It

requires knowledge of what peripherals remain active when processor goes into sleep mode. This was not implemented at this time, but it is considered for future work.

## 6. TESTING AND RESULTS

Testing of the functionality and efficiency of the overall system has been done. Yet, more extensive testing is to come, because some new functionalities should be done and test benches should be written.

The footprint of lwIP is about 20 Kbytes. This footprint is achieved with UDP and TCP included, plus DHCP and other functionalities. If functionalities of the stack, that are not needed, are excluded, then the footprint can be further reduced.

The footprint of the EFSL is about 12 Kbytes. This footprint can not be reduced further, because there are no functionalities that can be excluded.

FreeRTOS has a footprint of about 35 Kbytes. This footprint is with all options included, but basic configuration should not exceed 10 Kbytes.

At last, when the system has started functioning, additional test samples has been made, to test proper function of some of the functionalities. The simplest test that has been made was for testing if the user tasks are working correctly. This test was creating a task that toggles a LED on the development board. Of course after setting the right priority of this task, it worked as it should. The second test was more complicated. It was intended to test the interrupt functionality. Using interrupts under RTOS might be a little harder than on a system without it. With RTOS, interrupts can not be handled directly from C compiler, because the compiler is not aware of the operating system. This means that if interrupt is called and handled directly from C code, the compiler will not save the currently running task context, so, after returning from the interrupt, the task context will be messed up. This requires a separate assembler macro that will save the context and after that call the interrupt handler, normally written in C. After returning from the interrupt, the context is restored and the control is returned to the RTOS and currently running task. For testing interrupt handler functionality, two examples have been written. The first one was using timer interrupt and was toggling a LED on the development board. The second one was for using LPC2378 serial port. It had interrupts for sending and receiving data via serial port. The system was connected on a PC and tested with hyper terminal.

The next test was for the lwIP stack. First thing, was testing of the DHCP client. The system was connected on the network and DHCP server has been set up on a PC. The lwIP successfully obtained IP address from the server. Other test, concerning data sending and receiving has been also performed. For this test, special software for the PC has been developed. This software had a server, listening on a port and awaiting connection from the client. The client was the LPC2378 system. The client connects on the host machine and starts sending data. The server checks if that data is correct and measures the data rate. Achieved data rate was about 700 kbit/s, even the system was connected on 100 MB/s network. The low data rate is because of the low processing power of the microcontroller, but this data rate is more than enough for a lot of applications. This data rate was achieved using UDP. For TCP transfers, lower data rate is expected, because this protocol is more complex than UDP. Also, our system has been configured as a server, listening on a port. The system successfully accepted connections from other clients.

Finally, the EFSL has been tested. Only basic tests were performed in this case. An SD card of 2 GB has been used for this test. The test scenario has been including creation of file, sequential writes and reads and random writes and reads. These tests passed successfully.

## 7. FUTURE WORK

The future work will be mainly focused of writing drivers for the microcontroller peripherals (ADC, PWM, RTC, SPI, $I^2C$, CAN, USB etc.). The most of the peripherals would not require porting of new software, but the USB would require an USB stack. The stack is not yet chosen and probably some open source stack will be used.

Of course, additional tests should be written to test the lwIP and EFSL. Especially EFSL was not tested extensively, so extra tests should be performed, regarding functionality and speed.

## 8. LITERATURE


[1] FreeRTOS.ort, www.freertos.org, Bristol, UK

[2] Richard Barry, Using the FreeRTOS real time kernel, LPC 17xx edition, Real Time Engineers Ltd., 2010

[3] lwIP wiki, http://lwip.wikia.com/wiki/LwIP_Wiki

[4] lwIP official site http://savannah.nongnu.org/projects/lwip/

[5] Adam Dunkels, Design and implementation of the lwIP TCP/IP stack, Swedish Institute of Computer Science, February 20, 2001

[6] EFSL official site, http://efsl.be/

[7] C. (Kees) Pronk, Verifying FreeRTOS; a feasibility study, Report TUD-SERG-2010-042, Delft University of Technology, 2010