Romantiki OS – A single stack Multitasking Operating System for Resource Limited Embedded Devices.

Roman Glistvain[1], Mokhtar Aboelaze[2]
Dept. of Computer Science and Engineering
York University
Toronto, ON. Canada
[1]romangl@email.com
[2]aboelaze@cse.yorku.ca

***Abstract*-- Most resource limited embedded systems are programmed using super-loop architecture [21]. Although programs written for super-loop architecture are hard to debug and maintain, however it requires much less memory which makes it suitable for resource limited devices.**

In this paper, we propose an operating system "Romantiki" that combines the resource efficiency of super-loop architecture while featuring traditional multitasking coding style which makes it easy to develop and maintain complex projects for. We also compare the speed and memory requirements for Romantiki with a commonly used OS for embedded devices (FreeRTOS).

I. Introduction

Embedded processors are gaining popularity in many devices. The number of embedded processors sold is more than 20 times the number of general purpose processors in desktop PC's. The use of embedded processors ranges from TV and DVD players to toasters and microwave ovens. The average car contains more than 10 embedded processors for fuel injection, anti-lock brakes, and many other functions.

Low end embedded processors that are widely used in embedded systems and sensor networks are characterized by a simple processor with a limited processing power, limited resources, and low power consumption. They also have a low-to-moderate size memory without a Memory management Unit (MMU). These processors require special attention in developing application programs for. For example NXP LPC1111 has a 8K byte of programmable flash and 2K byte of SRAM. It is possible of course to get embedded processors with a more memory but that affects the price and power consumption of the device.

In this paper we propose a simple operating system that can run on embedded devices with very limited memory resources. The operating system is called Romantiki, which is Russian for "day dreamer", and it offers the feature-set of a standard cooperative multitasking operating system. Our design requirement for Romantiki is to satisfy five important criteria:

1) Fast startup time.
2) Small footprint in both RAM and Flash.
3) Fast response to time critical events. Thus, allowing the operating system to be used in Soft Real-Time systems.
4) Multitasking OS model for writing application code. Allows code reuse between resource limited embedded devices and devices running traditional operating systems.
5) "Socket-like" API to write networking applications. Thus providing the ability to create a common networking abstraction layer between Romantiki OS and traditional operating systems.

These requirements are met by placing tough coding guidelines on the developer. The initial learning curve of following those guidelines may result in a prolonged development cycle. However, the benefit derived by making the system use limited resources is far greater.

The remainder of the paper is organized as follows: Section II is a motivation for our work and describes Related Work. Section III describes local continuations. Section IV describes the overall structure of Romantiki OS. Section V provides benchmarks of running similar applications on Romantiki vs. FreeRTOS Operating System. Section VI is the direction of the future work. Section VII provides the conclusion which defines the goals achieved with the Romantiki Operating System.

II. Motivation and Previous work

A. Motivation

In his paper we focus on resource-constrained embedded devices which have network based user interface (UI). Those devices include network routers, managed switches, automation equipment, wireless sensor network devices, military surveillance, and communication devices. We will show that it is possible to write software for such devices following standard OS like architecture and create real-time software components which can be shared between complex devices and resource limited devices. There are many applications for our proposed operating system, these include:

1) Making existing systems cheaper. Various systems have a requirement for networking interface for

management purposes and currently employ complex microcontrollers and expensive operating systems. Using simple microcontrollers will reduce the cost of the system.

2) Systems which require very fast startup time, in order of milliseconds. One example of such device is an industrial automation Modbus/TCP station. To allow placement of those devices on a robot arm, the device needs to be operational within a very short period of time. Our proposed operating system is ideal for these situations.

3) Battery operated wireless devices. Many home automation wireless devices such as thermostats can benefit from having a network GUI to be remotely controlled.

One of the main advantages of our proposed operating system is that programs written for Romantiki enjoy a significant improvement in memory utilization compared to standard Operating Systems. In this paper we compare the memory requirements for running the same application code on a small preemptive operating system FreeRTOS and Romantiki OS. The RAM requirement for the FreeRTOS based application is about two times larger which means that the device needs to employ a bigger and less energy efficient CPU.

B. Related Work

Operating systems for embedded processors usually fall in one of two categories: run to completion systems and preemptive multitasking systems. In run to completion systems once a process is started, it will run until it is completed, then the next process is scheduled and so on. In preemptive multitasking a running job can be preempted to run a higher priority job. After the higher priority job is completed the preempted job could be resumed. A run to completion system requires less memory resources since only one process is running at any time, and all programs share the same stack. However, it may be hard to handle deadlines for real time jobs. The lack of preemption may result in missing important deadline because a low priority job is running and can not be preempted. On the other hand preemptive multitasking makes it easier to handle deadlines but it requires large amounts of memory since every task running or preempted must have its own stack [19]. Below is a brief description of previous work on operating systems for embedded applications.

Baker in [2] discussed the theoretical possibility of building a system capable of preemption using a single stack. In his work he assumed that the stack grows with every preemption and shrinks when a job is completed. He also assumed that resource requirements and deadlines are available offline. Although his work requires only a single stack, but it does not save any memory.

Traditional approaches of using the standard cooperating [14] or preemptive [10], [11] multitasking suffer from the problem of having to allocate separate stacks on a per task basis which are hard to calculate and overall result in an inefficient use of memory resources [12].

The concept of Y-Threads [13] makes it possible to create an operating system where the application programmer is responsible for identifying the preemptable and non-preemptable parts of the code. The preemptable portions are assigned private stacks, while the non-preemptable portions run on a common stack. The system allows application programmer to choose a tradeoff between performance and memory utilization. Even though, the use of common stack allows to provide significant improvements in memory resource efficiency, this system still requires estimation of multiple stacks which can be hard in many cases [12].

In Mtss, [12] the authors proposed a technique where tasks that need more stack space use stacks of other tasks thus reducing the memory requirements of the program.

Mantis [3] is a multitasking operating system designed specifically for sensor nodes in a sensor network. Mantis allocates individual stack space for tasks and then uses round-robin scheduling of tasks in the same priority level using time slicing.

There has been a significant amount of research in the area of limited memory embedded systems. In particular the recent research focuses on the area of Operating Systems with an integrated TCP/IP networking for small microcontrollers with embedded Flash and RAM [8], [10], [11], [14],[20].

Some operating systems such as Linux and Windows CE have an integrated TCP/IP stack and provide many standard features of their desktop counterparts. They have a very simple interface for the creation of processes and tasks without forcing the user to worry about the stack allocation for each task. This is accomplished by requiring the use of Memory Management Unit (MMU) hardware which provides protection mechanisms as well as a simple way of dynamically growing task stacks. Since, many embedded microcontrollers don't have the MMU hardware; they require a different kind of operating system, the kind which accomplishes the task of task scheduling with minimum amount of hardware support.

The work of Adam Dunkels [8], [6], [7]serves as a guideline that it is possible to have TCP/IP functionality as well as a small operating system running on small microcontrollers. His work on Protothreads [6], which provides a way to perform context switch using stack rewinding, laid the foundation for the Romantiki operating system. He developed Contiki [8] event-driven operating system which provides a feature-set similar to Romantiki OS. The main difference between Contiki OS and Romantiki OS lies in their target applications.

The Contiki Operating System with uIP stack is primarily targeted to 8 and 16 bit microcontrollers with very small amounts of RAM such as 2kb. Contiki doesn't have task priorities and therefore, it is hard to use in

applications where task level real-time response is desired.

The Romantiki operating system is targeted to new 32 bit microcontrollers with 16kb or more of RAM. It is designed as a cooperative multitasking operating system with task priorities. This design makes it possible to service certain types of real-time [15] events and therefore, it is targeted towards more complex embedded devices than Contiki OS. Moreover, the use of traditional inter-task communication and multitasking coding style makes it easy to provide code reuse and share the same codebase between traditional microprocessors and memory constrained microcontrollers.

III. Local Continuations

A. Overview

Local continuations [6] provide a way of creating multitasking functionality without creating separate stacks for individual tasks. Even though this functionality requires certain changes in the code structure compared to traditional preemptive multitasking, it provides a way of achieving an efficient use of memory resources. The program based on local continuation uses single stack to provide multitasking functionality.

Local continuations are used extensively in state machine based programs running on top of superloop systems - microcontroller applications that do not have an underlying operating system. Various local continuation libraries such as Protothreads [6] provide functionality which allows super-loop programs to be created using multitasking programming style. This provides the benefit of creating multitasking functionality which is easy to write and maintain without the overhead of an operating system.

B. Continuations

Traditional operating systems allow suspending and resuming running tasks by recording the current state of the task into the stack. Task state contains values of local variables, return addresses of nested functions and the set of CPU registers required to restore the execution of a suspended task.

Continuations provide a mechanism of saving and restoring the execution of a specific function. The state of the function contains state of CPU registers as well as the values of local variables. Regular continuations have the same memory requirements as the traditional operating system tasks since they need to save the local variables of functions on a function stack.

C. Local Continuations

Local continuations save only the execution state of the function and not its local variables. The execution state is called "continuation point" and typically contains the snapshot of CPU registers at the place in the function the execution needs to be resumed from.

This functionality is similar to state machine programs which maintain the state of execution as well as state related variables in the global memory. Local variables in state machine functions are used only for short time computations but are not maintained across multiple invocations of the state machine handler. Figure 1 show a fragment of code demonstrates a typical state machine function performing long running computation.

```
// state
unsigned long calc_st=CALC_ST_INIT;
// state variables
double calc=0;
unsigned long counter=0;
// state machine function
void calc_state_machine(double* calc, int* res)
{
    switch (calc_st)
    {
    case CALC_ST_INIT:
        counter=0;
        *calc=1.0;
        calc_st=CALC_ST_COMP;
        *res=CALC_IN_PROGRESS;
        break;
    case CALC_ST_COMP:
        if (counter<10)
        {
            *calc+=pow(*calc,3.5);
            counter++;
        }
        else
        {
            *res=CALC_DONE;
            calc_st=CALC_ST_INIT;
        }
        break;
    }
}
```

Fig. 1. A typical state machine based function.

Since state related variables in state machine programs are not saved on a stack, the use of stack in such functions is very limited. Moreover, since state machine programs run to completion, their memory requirements are much smaller than multitasking in preemptive operating systems.

Local continuations follow the same rules as state machines; therefore, multitasking functionality based on local continuations results in a significant memory saving compared to traditional multitasking operating system projects.

One major drawback of state machine based programming is the difficulty to create complex projects which are easy to understand and maintain. The example state machine code in Fig. 1 is hard to understand by simply looking at it. Local continuation libraries provide a way of hiding details of underlying state machine using standard C preprocessor.

The code in Fig 2 represents the same function as in Fig 1 rewritten using a simple macro library.

```
BFD cont_func(double* calc, int* res)
{
    static int counter;
    BF_BEGIN
    counter=0;
    *calc=1.0;
    *res=CALC_IN_PROGRESS;
    for (counter=0; counter<10; counter++)
    {
        *calc+=pow(*calc,3.5);
        BF_YIELD
    }
    *res=CALC_DONE;
    BF_END
}
```

Fig. 2. The same code in Fig 1 rewritten using macro library

The above code fragment is easy to read and understand with exception of a few API calls: BF_BEGIN, BF_YEILD and BF_END which are placed in different parts of the blocking function. Those API calls are preprocessor macros are shown in Figure 3.

It is evident that the code in Fig. 2 which appears after the macro expansion is a state machine code similar to Fig. 1. However, the code in Fig 2 is much easier to read and understand. This serves as a basis of using local continuations to express complex projects.

Local continuations allow writing multitasking code while maintaining a memory footprint of a state machine based program. This is accomplished by hiding details of state machines using standard C preprocessor.

```
enum
{
   BF_FINISHED,
   BF_BLOCKED
};

#define BFD char
#define BF_BEGIN    static unsigned short local_cont=0; \
                     switch(local_cont) \
                     { \
                              case 0:
#define BF_YIELD    local_cont=__LINE__; \
                     return BF_BLOCKED; \
                     case __LINE__:;
#define BF_END local_cont=0;} return BF_FINISHED;
#define BF_EXIT local_cont=0; return BF_FINISHED;
#define BFC(func) if ((func)==BF_BLOCKED) \
                       { \
                      BF_YIELD;\
                  }
```

Fig. 3. The macro needed for Fig. 2

The small macro library defined in Fig 3. makes it possible to implement nested blocking calls:

```
BFD block_func_nest_level_one()
{
    static int res;
    static double calc;
    BF_BEGIN
    res=CALC_IN_PROGRESS;

          BFC(cont_func(&calc,&res));

    printf("Result=%f\n",calc);
    BF_END
}
```

Fig. 4. Example of a non-reentrant blocking function

The above code in Figure 4 shows how it is possible to create nested non-reentrant blocking functions. This transformation is used in a number of continuation libraries [6], [4] and it serves as a basis for the continuation library of Romantiki operating system.

D. Protothreads

The protothread library[6] by Adam Dunkels uses local continuation functionality to provide multitasking coding style. The library provides various blocking constructs which makes it possible to construct complex multitasking projects. This was the first local continuation library which allowed the blocking calls to be made in nested continuation functions.

The protothread library is used in variety of applications such as Contiki operating system and uIP TCP/IP stack. It serves as the basis for the implementation of the continuation library used in the Romantiki operating system.

IV. ROMANTIKI OS

A. Overview

Romantiki OS is based on a classical multitasking model. The Romantiki OS consists of a kernel which provides various services and a task scheduler. The application project consists of multiple tasks each of which uses services provided by the kernel. Figure 5 shows a block diagram for the Romantiki operating system.

Task scheduling in Romantiki is cooperative which means that the scheduler will allow a high priority process to run only when the current task yields the control to the scheduler. The scheduler of Romantiki is based on a bitmap scheduler [16] with 31 tasks allowed in the system. Each task has a unique priority between 1 and 31.

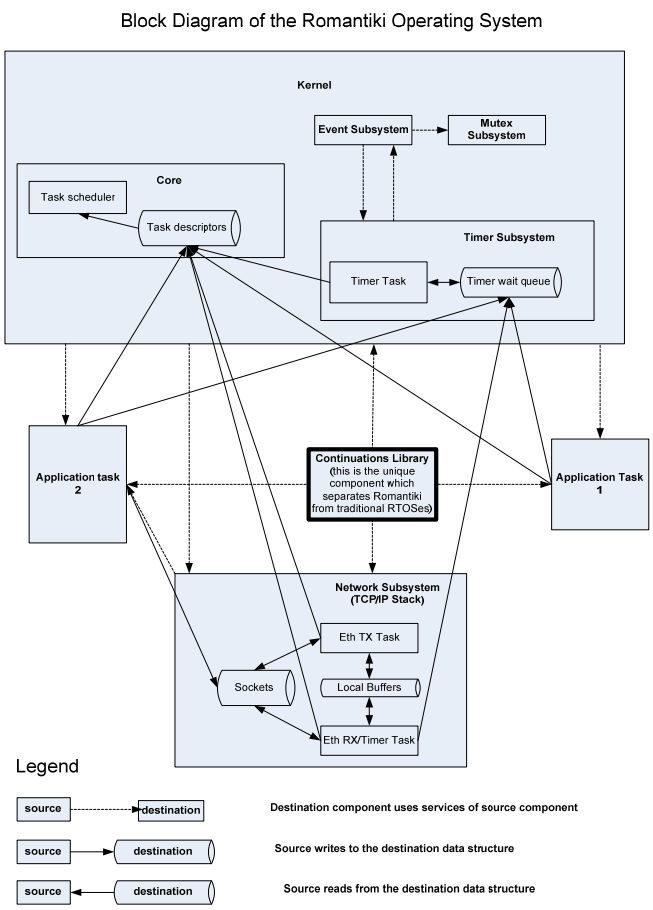

Fig 5. Romantiki Operating System

The uniqueness of the Romantiki OS is due to the use of local continuations to perform context switch.

B. *Local continuations library in Romantiki*

Romantiki OS uses local continuations to provide the functionality of blocking system calls. The library provides a number of features not present in other local continuation libraries. The features of Rmoantiki are summarized as:

1) Provides an abstraction layer which hides the definition and management of continuation points away from the developer. This makes the code easier to write and maintain.
2) Supports reentrant and non-reentrant blocking functions with different sets of macros.
3) Reduced functionality of the local continuation library. Event handling is part of the intertask communication API and is integrated into the operating system kernel.
4) The Romantiki OS local continuation library allows making blocking calls within nested functions which is standard practice in traditional operating systems. This makes it possible to adapt existing code into the Romantiki and maintain the same code-base between different operating systems. Other local continuation libraries integrated into real time operating systems such as Salvo RTOS[17] and FreeRTOS with CoRoutines [10] do not provide this functionality.

C. *System Calls*

The Romantiki Operating System implements the following kernel services:

1) Task manipulations: Task Creation, Task Startup, and Conditional Yielding
2) Events: Event creation and Event triggering. Both can be executed at a task level or interrupt level. Tasks blocks waiting for an event using `WaitSingleEvent` Tasks can also wait for multiple events using `WaitMultipleEvents`
3) Timer API: `OS_Sleep` blocks the running task for the given number of milliseconds, while `getOsTick` Gets the snapshot of the operating system tick counter.
4) Semaphores: implements both `semTake` and `semGive`.
5) TCP/IP Socket API: Provides TCP/UDP Interface.

The system call API is small compared to the traditional operating systems. The typical functionality of task deletion, suspension, dynamic memory management and message queues is not part of the kernel. It is possible to implement some of the above functionality in the application level based on the project requirements. However, many embedded systems can be implemented without the above functionality. Moreover, leaving this functionality outside the scope of the OS kernel makes it possible to create a simple operating system which is easy to learn and use in many embedded applications.

D. *Events in Romantiki*

The Romantiki operating system uses events as a main mechanism for inter-task communication and it builds more advanced objects such as timers, sockets and mutexes using events. In Romantiki, a task can wait for one or more events while other tasks or interrupts trigger events. The functionality of the event subsystem in Romantiki is similar to Event API in Windows [9]. The event API consists of 2 function groups:

1) Trigger Event. This is a non-blocking function which can be called from tasks and interrupts
2) Wait For Event(s). The calling task is suspended until one or more events it is waiting for have been triggered.

Each event in Romantiki has an event control block (ECB) associated with it. Since tasks in Romantiki can be blocked waiting for events, unrelated events should not unblock the waiting task. Therefore, each ECB contains a pointer to the Task Control Block (TCB) of the task which is currently blocked on the event. Other fields in the ECB contain event status and configuration information.

The event triggering function is part of the kernel. Therefore, it sets the task pending as soon as the event is triggered. The diagram below in Figure 6 demonstrates

the process of event handling and an interaction between an interrupt, scheduler and a task.

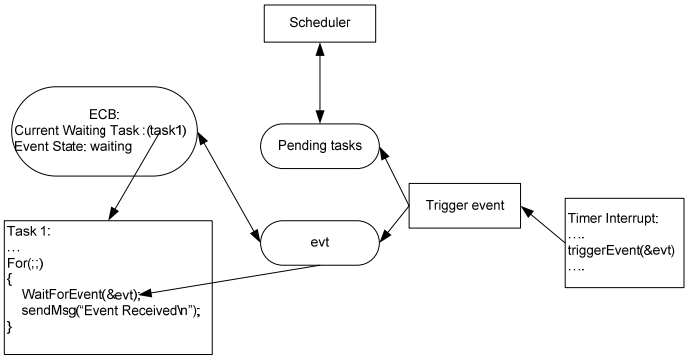

Fig 6. Interaction between events and the ECB

There is a limitation associated with events in Romantiki that only one task can be blocked on a certain event at a certain instance of time.

E. Tasks in Romantiki

In Romantiki a task can be in one of the 3 states: blocked, pending or running as shown in Fig. 7. The scheduler maintains a list of pending tasks. The pending list is updated during event triggering or conditional yielding. When the currently running process yields, the scheduler gets a chance to run. It selects the highest priority task from the pending list and invokes its processing function.

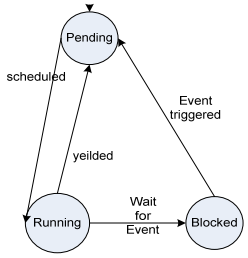

Fig 7. State diagram representing task state.

F. Conditional Yielding

Yielding is a common mechanism in cooperative operating systems to allow other tasks to run while one task executes long running operations such as complex arithmetic. During the process of yielding, the scheduler gets control and it selects the next task from the pending list. If there are no other tasks present in the pending list, the current task is re-invoked.

In an operating system where each task has a unique priority, a task should yield only if there is a higher priority pending task. Therefore, the use of an unconditional yield statement results in a wasted invocation of the scheduler and frequent rescheduling of the currently running task.

Romantiki operating system avoids the inefficiency of unconditional yielding by providing a construct which forces the task to yield only when higher priority task is pending. It is possible to provide this functionality since all the inter-task communication mechanisms are implemented using events and each task has a unique priority in the system.

This functionality is implemented by allowing the developer to insert special command "PREEMPTION_POINT" into his code. This command is a macro which contains the following functionality:

1) check global variable "preemption_request".
2) If "preemption_request==TRUE" then `YIELD`.
3) Otherwise continue execution of the current task.

The preemption_request variable is updated inside the event triggering function "`asyncTriggerEvent`" which checks whether or not the task waiting on the event has a higher priority than the currently running task.

Figure 8 provides an illustration of the conditional yielding operation. Each state transition is marked with a number which indicates the sequence of that specific event.

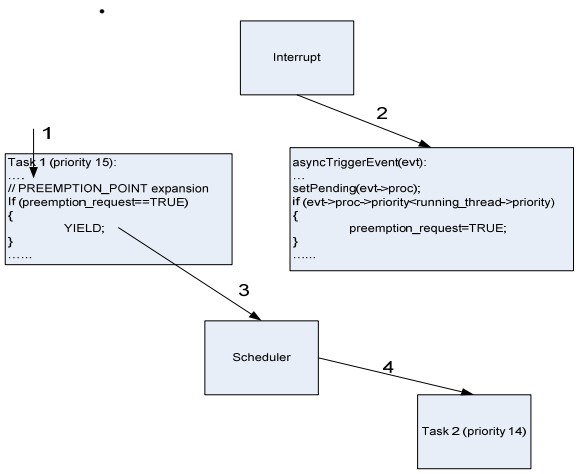

Fig 8. Conditional yielding

G. Real-Time application support

The real-time response can be accomplished on a cooperative basis if all the tasks do not run for a long time without yielding. This is accomplished by placing "PREEMPTION_POINT" macros across long running operations and various points in tasks.

H. Overhead of Romantiki OS

The Romantiki OS has a very small footprint. The core code of Romantiki requires 2.2 Kbytes of Flash and 316 bytes of RAM. This measurement was done when the code was compiled for AT91SAM7X256 ARM

microcontroller using GCC compiler 4.3.2 with optimization level –O2.

The core of the Romantiki OS consists of the task handling API, scheduler, timer subsystem and event subsystem.

I. Code Sharing.

The Romantiki operating system is unique because it is based on a traditional multitasking model while using a single rewinding stack for all tasks. This approach allows creating projects which have footprint of super-loop programs while the application code is easily maintainable, extendable and can be shared with projects running on preemptive operating systems.

The Romantiki OS uses a concept similar to Y-Threads where the developer needs to identify sections of the code which perform blocking calls and other sections which run-to-completion. This way, the developer needs to follow certain rules when defining blocking functions, while the non-blocking functions can use the unrestricted C language as they run to completion.

A common API can be created which is used by applications running on Romantiki OS and traditional operating systems. The goal of this API is to create transparent OS abstraction layer which would allow common operations such as Task definition and creation, Event API, and Timer API

The following are the details of the implementation of this abstraction layer in FreeRTOS preemptive operating system:

1) The task definition and creation is accomplished via FreeRTOS task API.
2) The task control block is extended to provide data storage required for the events associated with tasks
3) Tasks in FreeRTOS are assigned individual stacks.
4) Event API is implemented using binary semaphores provided by the FreeRTOS.
5) Timer API is implemented using FreeRTOS timer subsystem.

V. Performance Evaluation

In order to evaluate the Romantiki operating system, a typical embedded project was chosen and implemented using the same codebase under two operating systems Romantiki OS and FreeRTOS, a preemptive multitasking operating system. The test project implements an application where multiple protocols are multiplexed into a single communication channel. Many different applications are based on multiplexed channel model such as TCP/IP[7] DeviceNet[5] and MPEG-TS[22].

The test project runs on AT91SAM7X-EK[1] evaluation board and uses an Ethernet channel to implement multiple encapsulated protocols. The implemented encapsulation is much simpler than TCP/IP.

In this project a number of tasks and API calls are used to implement the basic encapsulation and message transmission. There are also a large number of application tasks that use the encapsulation services to transmit messages. Each client and server application is implemented as a task in the operating system. This approach results in a large number of tasks and a multiple context switches to transmit and receive messages. Figure 9 shows a block diagram of the test project.

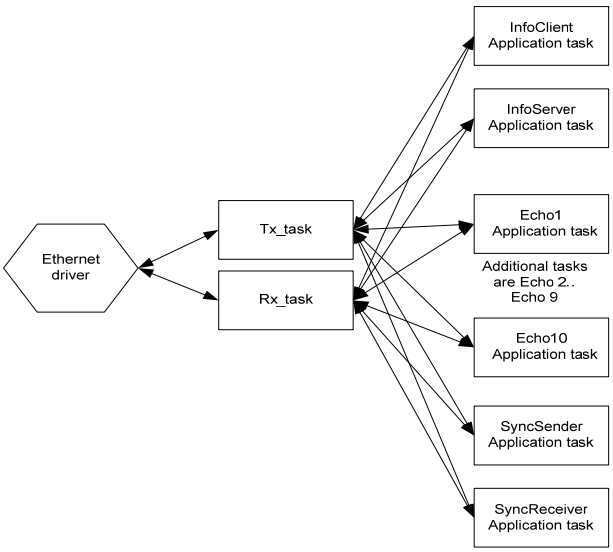

Fig 9. Test project for Romantiki

The test project contains 16 tasks which implement encapsulation and variety of application protocols. The performance of the different platforms was measured using the setup described Figure 10.

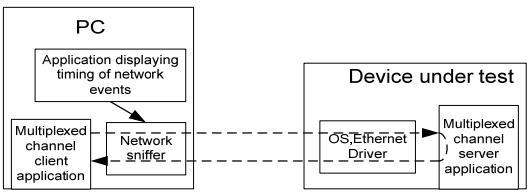

Fig 10. performance measurements for the test project.

The absolute timing of network frames was captured using the network sniffer and results were averaged over 5 samples for each platform.

Table I shows the memory requirements and the request response cycle of Romantiki and FreeRtos running on the hardware mentioned above.

TABLE I
A comparison between Romantiki and FreeRTOS

	Request response cycle µ sec	Flash size in KB	RAN size in KB
Romantiki	664.0	12.956	14.732
FreeRTOS	694.2	14.588	31.668

Even though both operating systems are quite similar in size and share an identical application code, there is still about 10% difference in size of the application which is primarily attributes to the simulation of Event API in FreeRTOS.

The biggest advantage of the Romantiki OS is the use of a single rewinding stack for all tasks. This eliminates the need to perform individual stack estimation where memory is commonly overestimated in order to accommodate future growth of the stack. In the FreeRTOS test project each application task is assigned its own 1 kb stack. In Romantiki a single 1kb stack is used for all tasks and additional storage within the task descriptor which is used to store variables of reentrant blocking functions. Since there are 16 tasks in the system the stack allocation represents a significant amount of memory resources on the FreeRTOS platform. The reason for the large stack in each task is recursive functions and callbacks which run to completion under Romantiki but can be preempted in FreeRTOS.

The Romantiki OS provides 54% improvement in the memory use (RAM) for the test application compared to the same application running on FreeRTOS. Also the system is slightly faster under Romantiki.

It is possible to avoid using the overestimation strategy and tweak the stack size for individual tasks on FreeRTOS. This will reduce the memory use in FreeRTOS application by a large margin. However it will increase the development time and make it hard to find problems in the system since for each software change, stack sizes need to be reevaluated. In Romantiki OS overestimation is performed over a single stack while in FreeRTOS it is performed over multiple stacks and this causes the large difference in memory use.

VI. Future work

The future work will concentrate on implementing TCP/IP functionality in Romantiki. We will also add preemptive multitasking support for Romantiki based on SST [20].

VII. Conclusion

Romantiki was designed as a proof of concept of a multitasked operating system which fits the RAM and Flash footprint of corresponding superloop project. It is very compact and easily portable to different architectures.

When the number of application tasks is high, the Romantiki OS provides very significant improvement in RAM efficiency compared to traditional preemptive operating systems.

This operating system allows constructing of complex devices using very small and cheap microcontrollers.

The application code written for Romantiki can be recompiled to a traditional preemptive operating system such as FreeRTOS without making any changes to the application. This is accomplished by using a common OS abstraction layer when writing application code.

References.

[1] AT91SAM7X-EK Evaluation board http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3759. Accessed Jan. 2010.

[2] T. P. Baker. “A stack-based resource allocation policy for realtime processes”. *Proc. Real-Time Systems symposium*, 1990 pp 191–200.

[3] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. “Mantis os: an embedded multitasked operating system for wireless micro sensor platforms”. *Mobile Networks Applications*, 10(4) pp563–579, 2005.

[4] S. Tatham. (2010) CoRoutines in C [Online] Available http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html.

[5] (2010) DeviceNet Network [Online] http://www.ab.com/en/epub/catalogs/12762/2181376/214372/1768364/3404052/.

[6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali ”Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems” *Proc. of the 4th International Conference on Embedded Networked Sensor Systems* 2006. pp 29-42..

[7] A. Dunkels “Full TCP/IP for 8-bit architecture” *Proc. the 1st international conference on Mobile systems, applications and services*. 2003. pp 85-98.

[8] A. Dunkels, B. Grönvall, and T. Voigt “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors”. Proc. the 29th *Annual EEE International Conference on Local Computer Networks* 2004. pp 455-462.

[9] (2010) Event API in Windows [Online] Available http://msdn.microsoft.com/en-us/library/ms686915(VS.85).aspx.

[10] (2010) FreeRTOS Operating System [Online] Available http://www.freertos.org/

[11] J. labrosse *MicroC/OS-II: The Real-Time Kernel*: 2nd ed. 2002.

[12] B. Middha, M. Simpson, and R. Barua. Mtss: “multi task stack sharing for embedded systems”. *proc. the international conference on Compilers, architectures and synthesis for embedded systems*, 2005. pp 191–201

[13] C. Nitta, R. Pandey, and Y. Ramin. “Ythreads: Supporting concurrency in wireless sensor networks”. proc. *Intl Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2006

[14] (2010) Nut/OS [Online] Available http://www.ethernut.de/en/software/index.html.

[15] C. Penumuchu *Simple Real-Time Operating System: A Kernel Inside View for a Beginner*, Trafford Publishing 2007.

[16] (2010) Protothreads with Scheduler [Online] Available http://code.google.com/p/protothread/

[17] (2010) Salvo RTOS [line] Available http://www.pumpkininc.com/.

[18] (2010) eCos kernel overview [Online] Available http://ecos.sourceware.org/docs-2.0/ref/kernel-overview.html.

[19] A. Singhania, S. han, and M. Srivastava, “Knots: An efficient single stack preemption mechanism for resource constrained devices”. Networked and Embedded Systems Laboratory, UCLA, TR-UCLA-NESL-200710-02 October 2007.

[20] (2010) Super Simple Tasker [Online] Available http://www.embedded.com/columns/technicalinsights/190302110?_requestid=313577

[21] D. Stonier-Gibson (2010) “Understanding embedded microcontroller multitasking RTOS alternatives” [Online] Available http://www.splatco.com/rtos_1.htm

[22] P.A. Sarginson (2010) MPEG-2 Overview of systems layer [Online] Available http://downloads.bbc.co.uk/rd/pubs/reports/1996-02.pdf.