Embedded Real Time Operating System Using *C++*

Gautam Ganapati Hegde[1], V. Natarajan[2]

[1,2]*Department of Electronics and Communication Engineering, SRM University*

Abstract—**Following the current popular trend in industry it is found that most of the applications are written using object oriented programming language hence emphasizing on the advantages these languages hold over the structural or other kinds of programming languages. Such a trend is yet to be found in embedded field. Hence to test the untapped potential of use of object oriented properties in embedded world, we develop a Real Time Operating System using C++ as the selected object oriented language.**

Keywords—RTOS, Priority, Task, Scheduling, C++, Object Oriented

I. Introduction

A Real time operating system is a created environment where we are able to predict or determine the output and the time taken to get output from input consistently throughout the life of the system. Currently there are many RTOS being used in the world, where maximum of them whether open source or licensed or custom in-house developed are developed using C language [1]. For many years, C++ has been seen as the natural successor and has found greater acceptance, but the increase in its usage has been much slower than anticipated. There are various reasons for this like embedded developers are quite conservative and prefer to use solutions that are proven rather than novel, there is also the lesson of experience of failure of Embedded C++.

In the past Embedded C++ (EC++) language was developed which was a subset of C++ language. But EC++ had a lot of features missing from original C++ to decrease the code size and maximize efficiency and make compiler construction simpler. Due to stripped down version of C++ and lack of powerful micro controllers during that time lead to the failure of Embedded C++.

In recent times the scope for using of object oriented languages in embedded world has improved due to various factors such as: Better hardware where the memory and processing power has considerably increased while decreasing the size and consumption of power of micro controllers and developing of compiler support for variety of languages. Even popular RTOS such as FreeRTOS [2] has released limited support for C++ developed applications.

Hence seeing this recent trend, we design and implement a RTOS built using only C++ and a bit of assembly language while using the object oriented features such as objects, encapsulation and inheritance. This experiment is being done to observe the effects of using these properties and check if we have enough facilities available to improve the applications in embedded world.

II. DESIGN AND IMPLEMENTATION

This RTOS is being developed with very limited features using GNU GCC compiler [4] to observe its behavior and compare performance with RTOSs available in market before testing it with fully loaded features.

Features being implemented:

- Tasks have only 3 states that are Ready, Running and Wait/Waiting.
- Two types of scheduling algorithm: Preemptive Static Priority Scheduling and Cooperative Scheduling.
- Can have maximum of 16 tasks including an idle task.
- Two types of inter task communication available: Event flags and Mutex.

III. WORKING CONCEPT

A. *Tasks:*

In this RTOS developed using C++, the tasks which will be defined by the RTOS user will be an object each. Task Control Blocks (TCB) are not specially created for tasks. During the creation of these tasks the user will have to give it the priority value, stack size and state of the task. The priority values will be ranging from 1 to defined limit in configuration file (not more than 15) with 1 being highest priority. The stack size value given will be in terms amount of bytes to be reserved for the size of stack for each task. The value of state of task will be either 0 - not ready or in wait state, 1 – ready state and 2 – running state.

Few rules to be followed are: The number of tasks that will be used in the application needs to be defined in the configuration file and in main application code the number tasks must match that number. The number of tasks cannot be more or less than the defined value in configuration. In the range of tasks being defined priority value cannot be skipped i.e. if defined value of maximum tasks in configuration file is 5 then values set can only be from 1-5 without repetition. Of all the tasks being defined can have the state value set as 2 i.e. running, since multiple tasks cannot be run simultaneously by the processor. In case of using cooperative scheduling, no need to set the priority values during task creation.

B. *Scheduling:*

The priority values provided during task creation will be used as array index. There are two arrays – first for saving the state of all the tasks and second for saving the stack pointer to the task. The values will be stored in the array corresponding to the priority value defined by user. The scheduler will be using these arrays to schedule. Every time the scheduler is executed, a loop is run in which the scheduler checks which task is ready (state value 1) from highest priority value to lowest priority value. In worst case scenario no task will be in ready state in which case the loop will reach the last place in array reserved for idle task which will always have state value as 1 (ready state).

The scheduling algorithm for preemptive static priority scheduling is as such:-

- Start a loop
- Check values of state array from highest priority (value 1)
- When encountered with value 1 in status array, update the current task's stack pointer value in address array.
- Take the stack pointer value from address array for next task's execution.
- In case the highest ready task is the same task which was already running (value 2 in status array), the same task continues with execution.
- Every time scheduler is run, the loop is started fresh.

The scheduling algorithm for cooperative scheduling is as such:-

- Start a loop. This loop is forever running i.e. unlike in preemptive where a fresh loop is started every time, the loop is continued from previous step.
- Check which task in status array is ready to run.
- When found the next task, save the stack pointer of previous running task in address array.
- Load the next task to run address from the address array to run on processor.

C. *Inter task communication:*

There are two types or inter task communication being provided:

1) *Event flags*:

Event flags will be used for event to task synchronization. Each of these flags created will have an array to save the data regarding to which task is waiting for the flag. In the array, value 1 on the index location like status array represent that the task is waiting for that flag, value 0 means the task is not interested in the flag. Every time a task finishes using the flag, a loop is executed on the array to find the next waiting task for this flag. Every time wait function or signal function is executed, the array is updated with status of task regarding the flag and scheduler is executed. Wait() is a function used to make the task wait for an event indefinitely or to wait for certain amount of time before rechecking if it can continue its work. Signal() function is used to signal other tasks regarding to occurrence of an event due to that task hence causing an interrupt to check and cause state transition leading to execution of scheduler.

2) *Mutex:*

Mutex are binary semaphores used for protecting of common resources between multiple tasks. Mutex is used to lock the access to the resource by the task until the work with that resource is completed. All mutexes created will have an array of its own to refer the tasks waiting for the mutex. The values in array corresponding to the index location refer to the state of task regarding that mutex. Value 0 indicates that the task is not waiting for that mutex, value 1 indicates that the task is waiting for that mutex and value 2 indicates that this task has locked this mutex. Functions such as lock() is used by the task to try and lock onto the mutex if it is available. If it is not available the user can determine beforehand if the task should wait indefinitely until mutex is free or check condition if free after every timeout(value set by user). Function unlock() is used to unlock / release the mutex and hence the resource.

IV. IMPLEMENTATION OF THE CONCEPT

The status array and address array will be declared as global variables. Though use of global variable is considered risky and hence not preferred for use, in this case use of global variable is a good option because only select few functions can modify it and it helps in eliminating need of complicated data structures and TCB.

There are three main parent classes which will be for kernel and a base class for tasks. A class is defined for kernel which will contain all kernel level operations such as task management, scheduling at main program level and ISR level, inter task communication support and system timer support. It will contain functions such as:

- Schedule () : used to perform scheduling.
- Run () : used to start and run the RTOS.
- Get_tick_count () : to get current timer tick count value.
- System_timer () : used to access timer related functions.

Second parent class defined is base class for task which handles creation of tasks, initializing the global variable arrays (status and address array), initialization of stacks and task management functions such as :

- Set_task_ready () : used to set a task ready.
- Set_task_unready () : used to set a task unready / wait.

Two classes are derived by these parent classes to gain access to internal functions and data members by the user. This way information sensitive for working of RTOS are protected from the user.

Third parent class is created for inter task communications. This class contains few functions to access kernel level operations. Event flag and mutex are derived classes from this class and contain the implementation of functions related to their working.

Working of stack, interrupt service routines, critical section etc are defined and implemented in port files since they are heavily dependent on the type of target hardware with a mix of C++ and assembly language.

V. RESULTS

Having ported the RTOS on LPC2148 [5] we have successfully tested that the RTOS works, however due to lack of many features the performance cannot be yet compared against the RTOSs available in market. The applications built had to be very simple since only two types of inter task communication is available and only two types of scheduling which in this case is not enough to find much difference in performance between them. For the simple applications developed, performance is rather very similar to various RTOS available in market as it was found that minimum task switching latency at 50MHz is 7 us.

VI. CONCLUSION

Since the RTOS lacks a lot of features, more inter task communication features are being added after which an application can be developed to check the performance and compare the performance of same application on the RTOSs available in market on the same hardware. We will be able develop applications using Object Oriented Programming languages to run on embedded platform where its features such as encapsulation, inheritance, polymorphism etc can decrease workload on developers and increase code reusability by using programming models used in computer software world.

REFERENCES

[1] Nguyen Bao Anh, Su-Lim Tan, “Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers”, IEEE Micro “unpublished”.

[2] ‘FreeRTOS homepage’. Available at http://www.freertos.org/.

[3] Charles Crowley, “Operating System: A Design Oriented Approach”.

[4] ‘GCC homepage’. Available at https://gcc.gnu.org/.

[5] www.nxp.com/documents/data_sheet/LPC2141_42_44_46_48.pdf, ‘Datasheet of LPC 2148’