International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 1 Issue 6, August - 2012

Porting & Implementation of features of µC/OS II RTOS on Arm7 controller LPC 2148 with different IPC mechanisms

Prof. Nilima R. Kolhare,
Asst.Prof,.Government College of Engineering,
Aurangabad, Maharashtra

Mr. Nitin I.Bhopale,
Lecturer, SRES College Of Engineering,
Kopargaon, Maharashtra.

Abstract- This paper describes an embedded system based on µC/OS II operating system using ARM7. It deals with the porting of MicroC/OS-II kernel in ARM powered microcontroller for the implementation of features like multitasking, time scheduling, mailbox and, mutex. Here a real time kernel is the software that manages the time of a micro controller to ensure that all time critical events are processed as efficiently as possible. Different interface modules of ARM7 microcontroller like UART, ADC and LCD are used. Data acquired from these interfaces is tested using µC/OS-II based real time operating system. It mainly emphasizes on the porting of µC/OS-II. It also shows how different applications are quite handy to use the RTOS features. With respect to the applications or tasks, this paper explains additional features of µC/OS-II like mutex, semaphore which are not inbuilt along with in built functions like multitasking, scheduling, mailbox.

Index Terms- embedded system, µC/OS-II, ARM7, RTOS

I. INTRODUCTION

In high end applications, sometimes devices may malfunction or totally fail due to long duration of usage or any technical problem which give fatal results. An embedded system is necessary for continuously collecting values from onsite and later analyzing that as well as taking proper measures to solve the problem. The systems that are in use today use non real time operating systems based on mono-task mechanism that hardly satisfies the current requirements. This paper will focus on porting of µC/OS-II in ARM7 controller that performs multitasking and time scheduling. The µC/OS II features and its porting to ARM7 are discussed. Finally it provides an overview for design of embedded system using µC/OS II and with respect to the response of the application, different features can be implemented. µC/OS II (pronounced "Micro C O S 2") stands for Microcontroller Operating System Version 2 and can be termed as µC/OS-II or uC/OS-II),. It is a very small real-time kernel with memory footprint is about 20KB for a fully functional kernel and source code is about 5,500 lines, mostly in ANSI C. Its source is open but not free for commercial usages. uC/OS-II is upward compatible with µC/OS V1.11 but provides many improvements, such as the addition of a fixed-sized memory manager; task deletion and deletion, task switch, Task Control Block(TCB) extensions support.

II. UC/OS II USING ARM

µC/OS-II, The Real-Time Kernel is a highly portable, ROMable, scalable, pre emptive real-time, multitasking kernel (RTOS) for microprocessors and microcontrollers. µC/OS-II can manage up to 250 application tasks. µC/OS-II runs on a large number of processor architectures and ports. The vast number of ports should convince that µC/OS-II is truly very portable and thus will most likely be ported to new processors as they become available. µC/OS-II can be scaled to only contain the features you need for your application and thus provide a small footprint. Depending on the processor, on an ARM (Thumb mode) µC/OSII can be reduced to as little as 6K bytes of code space and 500 bytes of data space (excluding stacks). The execution time for most of the services provided by µC/OS-II is both constant and deterministic. This means that the execution times do not depend on the number of tasks running in the application.

A. Choosing µC/OS II

µC/OS II is chosen for the following features:

1. Portable

Most of µC/OS-II is written in highly portable ANSI C, with target microprocessor specific code written in assembly language. Assembly language is kept to a minimum to take µC/OS-II easy to port to other processors. Micro C/OS-II can be ported to a large number of microprocessors as long as the microprocessors provides a stack pointer and the CPU register can be pushed onto and popped from the stack. µC/OS-II can run on most 8, 16, 32 or even 64 bit microprocessors or microcontrollers and DSPs.

2. ROMable

µC/OS-II was designed for embedded application. This means that if you have the proper tool chain (i.e. C compiler, assembler and linker/locater), you can embed Micro C/OS-II as part of a product**.**

3. Scalable

µC/OS-II is designed such a way so that only the services needed in the application can be used, means that a product can use just a few µC/OS-II services. This allows to reduce the amount of memory (both RAM and ROM) needed by µC/OS-II on a per product basis. Scalability is accomplished with the use of conditional complication.

4. Pre emptive

µC/OS-II is a fully pre emptive real time kernel. This means that Micro µC/OS-II always runs the highest priority task that is ready.

5. Multitasking

Multitasking is the process of scheduling and switching the CPU between several tasks. µC/OS-II can manage up to 64 tasks.

6. Deterministic

Execution time of all µC/OS-II functions and services are deterministic. This means that one can always know how much time µC/OS-II will take to execute a function or a service.

7. Robust and Reliable

µC/OS-II is based on µC/OS which has been used in hundreds of commercial applications. µC/OS-II uses the same core and most of the same functions as µC/OS yet offers more features.

B. Starting µC/OS-II

In any application µC/OS-II is started as shown in the figure 1.Initially the hardware and software are initialized. The hardware is the ARM core and software is the µC/OS-II. The resources are allocated for the tasks defined in the application.

Then the scheduler is started and it alligns tasks in pre-emptive manner. All these are carried out using specified Functions defined in µC/OS-II.

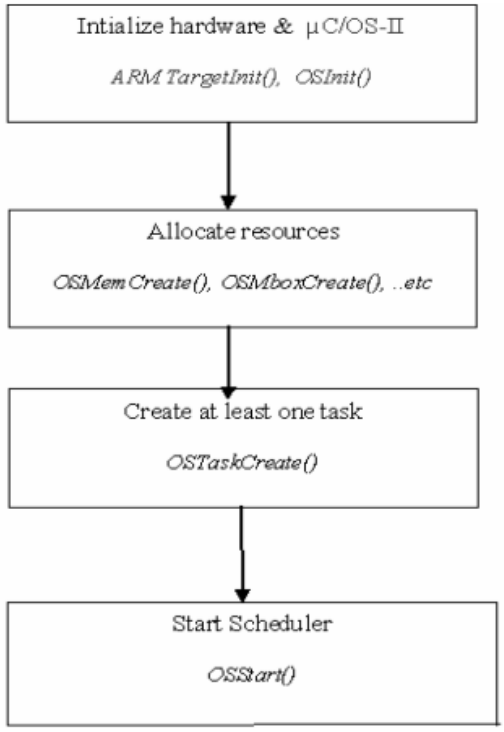

Figure 1: Starting µC/OS-II

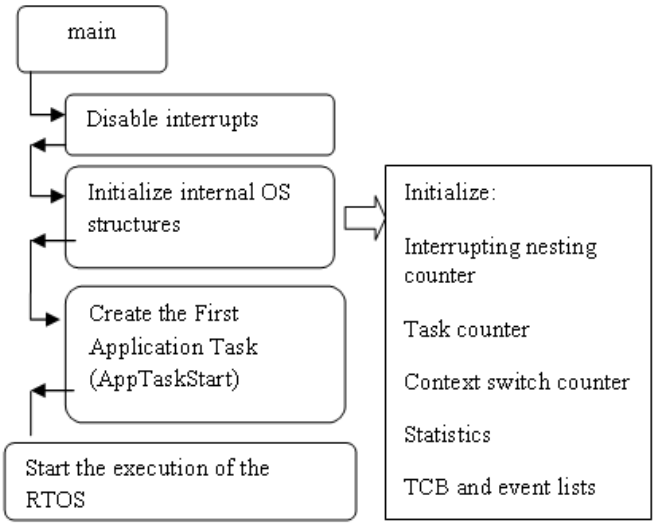

Figure 2: Initializing µC/OS-II

C. Initializing µC/OS II

µC/OS-II can be initialized as shown in the figure 2.As shown below the sample program to correlate steps shown above.

```
void main (main)
{
/* user initialization */
OSInit(); /* kernel initialization */
/* Install interrupt vectors */
/* Create at least 1 task (start task) */
/* Additional User code */
OSStart(); /* start multitasking */
}
```

D. Task Creation in µC/OS II

To make it ready for multitasking, the kernel needs to have information about the task: its starting address**,** top-of-stack (TOS)**,** priority**,** arguments passed to the task**.** Create the task before start of the multitasking (at initialization time)

```
OStaskCreate(void (*task) (void *parg),
void *parg,
OS_STK *pstk,
INT8U prio);
```

Mutual Exclusion

The easiest way for tasks to communicate with each other is through shared data structures. This is especially easy when all the tasks exist in a single address space. Tasks can thus reference global variables, pointers, buffers, linked lists, ring buffers, etc. While sharing data simplifies the exchange of information, by ensuring that each task has exclusive access to

the data to avoid contention and data corruption. The most common methods to obtain exclusive access to shared resources are:
a) Disabling interrupts
b) Test-And-Set
c) Disabling scheduling
d) Using semaphores

Mutual Exclusion, Disabling and enabling interrupts

The easiest and fastest way to gain exclusive access to a shared resource is by disabling and enabling interrupts as shown in the pseudo-code below

Disable interrupts;
Access the resource (read/write from/to variables);
Reenable interrupts;

Mutual Exclusion, Disabling and enabling the scheduler

If task is not sharing variables or data structures with an ISR then disable/enable scheduling, *Locking and Unlocking the Scheduler*) as shown in listing (using µC/OS-II as an example). In this case, two or more tasks can share data without the possibility of contention. While the scheduler is locked, interrupts are enabled and, if an interrupt occurs while in the critical section, the ISR will immediately be executed. At the end of the ISR, the kernel will always return to the interrupted task even if a higher priority task has been made ready-to-run by the ISR. The scheduler will be invoked when **OSSchedUnlock()** is called to see if a higher priority task has been made ready to run by the task or an ISR. A context switch will result if there is a higher priority task that is ready to run. Although this method works well, Mostly it should be avoided to disable the scheduler because it defeats the purpose of having a kernel in the first place.

Mutual Exclusion, Semaphores

The semaphore was invented by Edgser Dijkstra in the mid 1960s. A semaphore is a protocol mechanism offered by most multitasking kernels. Semaphores are used to:
a) control access to a shared resource (mutual exclusion);
b) signal the occurrence of an event;
c) allow two tasks to synchronize their activities.

A semaphore is a key that selected code acquires in order to continue execution. If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner. In other words, the requesting task says: "Give me the key. If someone else is using it, I am willing to wait for it!" There are two types of semaphores: *binary* semaphores and *counting* semaphores. As its name implies, a binary semaphore can only take two values: **0** or **1**. A counting semaphore allows values between **0** and **255**, **65535** or **4294967295**, depending on whether the semaphore mechanism is implemented using 8, 16 or 32 bits, respectively. The actual size depends on the kernel used. Along with the semaphore's value, the kernel also needs to keep track of tasks waiting for the semaphore's availability.

There are generally only three operations that can be performed on a semaphore: INITIALIZE (also called *CREATE*), WAIT (also called *PEND*), and SIGNAL (also called *POST*).The initial value of the semaphore must be provided when the semaphore is initialized. The waiting list of tasks is always initially empty. A task desiring the semaphore will perform a WAIT operation. If the semaphore is available (the semaphore value is greater than **0**), the semaphore value is decremented and the task continues execution. If the semaphore's value is **0**, the task performing a WAIT on the semaphore is placed in a waiting list. Most kernels allow you to specify a timeout; if the semaphore is not available within a certain amount of time, the requesting task is made ready to run and an error code (indicating that a timeout has occurred) is returned to the caller. A task releases a semaphore by performing a SIGNAL operation. If no task is waiting for the semaphore, the semaphore value is simply incremented. If any task is waiting for the semaphore, however, one of the tasks is made ready to run and the semaphore value is not incremented; the key is given to one of the tasks waiting for it. Depending on the kernel, the task which will receive the semaphore is either:
a) the highest priority task waiting for the semaphore, or
b) the first task that requested the semaphore (First In First Out, or FIFO).

Some kernels allows to choose either method through an option when the semaphore is initialized. µC/OS-II only supports the first method. If the readied task has a higher priority than the current task (the task releasing the semaphore), a context switch will occur (with a pre-emptive kernel) and the higher priority task will resume execution; the current task will be suspended until it again becomes the highest priority task ready-to-run. Figure 3 shows how you can share data using a semaphore (using µC/OS-II). Any task needing access to the same shared data will call **OSSemPend()** and when the task is done with the data, the task calls **OSSemPost()**.Semaphore is an object that needs to be initialized before it's used and for mutual exclusion, a semaphore is initialized to a value of **1**. Using a semaphore to access shared data doesn't affect interrupt latency and, if an ISR or the current task makes a higher priority task ready-to-run while accessing the data then, this higher priority task will execute immediately.

```
OS_EVENT *SharedDataSem;
void Function (void)
{
INT8U err;
OSSemPend(SharedDataSem, 0, &err);
. /* Access shared data here (interrupts are recognized) */
OSSemPost(SharedDataSem);
}
```

Accessing shared data by obtaining a semaphore.

Semaphores are especially useful when tasks are sharing I/O devices. If two tasks were allowed to send characters to a printer at the same time. The printer would contain interleaved data from each task, if task #1 tried to print

"I am task #1!" and task #2 tried to print "I am task #2!" then the result will be:I Ia amm t tasask k#1 #!2!

In this case, semaphore can be used and initialize it to **1** (i.e. a binary semaphore). The rule is simple: to access the printer each task must first obtain the resource's semaphore. Figure 3 shows the tasks competing for a semaphore to gain exclusive access to the printer. Note that the semaphore is represented symbolically by a key indicating that each task must obtain this key to use the printer.

This example implies that each task must know about the existence of the semaphore in order to access the resource. There are situations when it is better to encapsulate the semaphore. Each task would thus not know that it is actually acquiring a semaphore when accessing the resource. For example, an RS-232C port is used by multiple tasks to send commands and receive responses from a device connected at the other end of the RS-232C port. A flow diagram is shown in Figure 4.

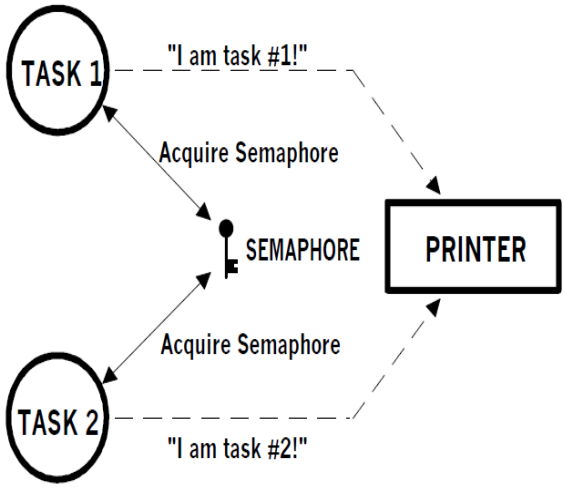

Figure 3: Semaphore for accessing printer

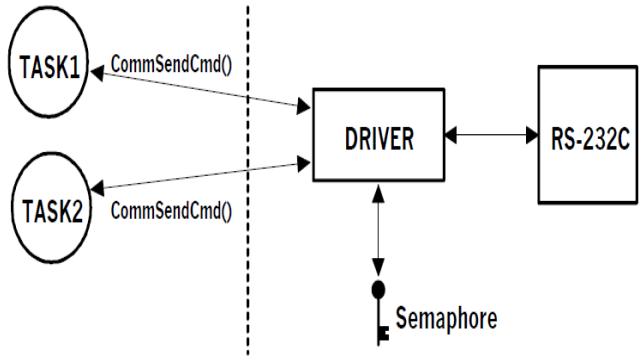

Figure 4: Hiding a semaphore from task

The function **CommSendCmd()** is called with three arguments: the ASCII string containing the command, a pointer to the response string from the device, and finally, a timeout in case the device doesn't respond within a certain amount of time. The pseudo-code for this function is:

```
INT8U CommSendCmd(char *cmd, char *response, INT16U
timeout)
{
Acquire port's semaphore;
Send command to device;
Wait for response (with timeout);
if (timed out) {
Release semaphore;
return (error code);
} else {
Release semaphore;
return (no error);
}
}
```

Encapsulating a semaphore.

Each task which needs to send a command to the device has to call this function. The semaphore is assumed to be initialized to **1** (i.e., available) by the communication driver initialization routine. The first task that calls **CommSendCmd()** will acquire the semaphore and thus proceed to send the command and wait for a response. If another task attempts to send a command while the port is busy, this second task will be suspended until the semaphore is released. The second task appears to have simply made a call to a normal function that will not return until the function has performed its duty. When the semaphore is released by the first task, the second task will acquire the semaphore and will thus be allowed to use the RS-232C port. Further the counting semaphore with buffer pool can be used

The heart of the system is a real-time kernel that uses preemptive scheduling to achieve multitasking on hardware platform. The previous sections dealt with µCOS_II porting to the application desired. This section deals with the implementation of hardware and software.

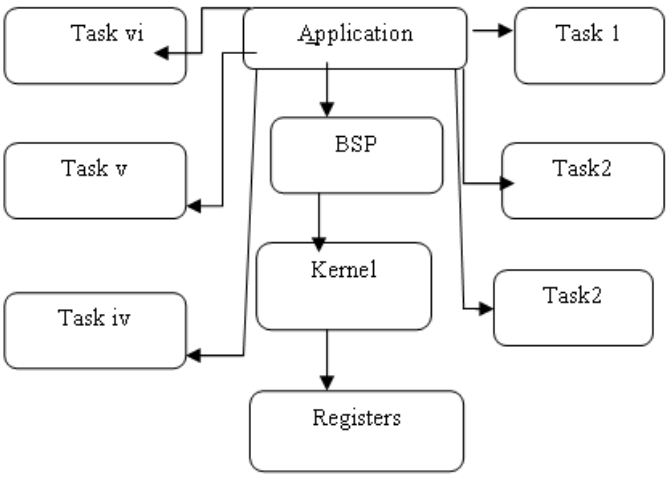

Figure 5: Block diagram of hardware platform

In Micro C/OS-II maximum number of tasks is 64. In the figure shown above the application has six tasks. Depending on the required application the number of tasks may vary. To perform a sample experiment to understand the porting of µC/OS-II we can perform simple tasks like Temperature sensor (i.e., ADC), Graphical LCD (i.e., degree to graphical Fahrenheit), UART (i.e., digital data displaying), LED toggle (ie., 8-bit data flow control) Buzzer (i.e., alarm device). The ARM runs the Real time operating system to collect information from the external world.Here RTOS is used to achieve real time data acquisitions. MicroC/OS-II kernel is ported in ARM powered microcontroller for the implementation of multitasking and time scheduling as shown in previous sections.

WinARM is used for implementation. WinARM is a windows operating system software program that runs on a PC to develop applications for ARM microcontroller and digital signal controller. It provides a single integrated environment to develop code for embedded microcontroller.

PROPOSED WORK

As shown in the figure 7 .Initialisation of the task1 and task2 is been done. Although supports for total 64 tasks all of them are not used at a time in application therefore with respect to demand the task must be created. These two tasks will acquire the values from the inbuilt ADC through channels 6 and channel 7.As soon as the values has been taken from ADC semaphore will be acquired by the tasks 1 and 2.The data to be sent to the hyper Terminal is to be converted firstly into the ASCII by Hex to ASCII conversion at the intermediated stage. Once it is sent to the hyper terminal semaphores acquired by the tasks will be released and in order to have the continuous check for all the above process the delay of 1 second is taken and all the will be repeated. In order to focus on the features of RTOS ,As shown in the figure task 3 is been reserved for the keypad application, similar to that the task 4 is been reserved with the LCD application.

As soon as the task 3 is activated at the same time task5 will be triggered which will simply implement the mutex by semaphore. In the next task5 will communicate to task 4, Value receives through the task 3 it simply display it on to the LCD at the same time it is also checked that whether the pressed key is the key 6 or key 7 accordingly the current value related to it will be transferred to the hyper terminal.

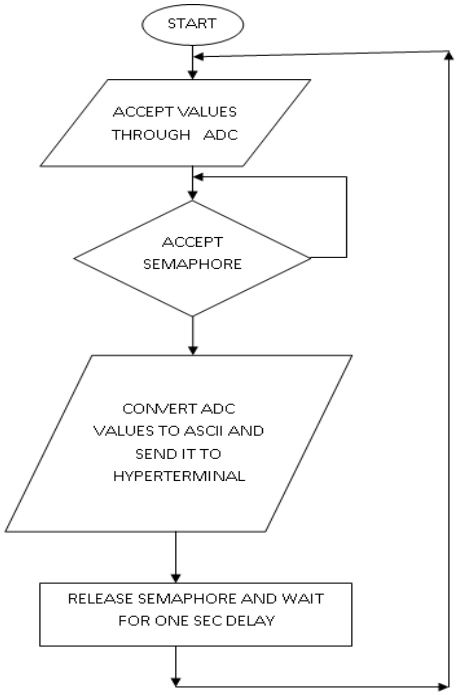

Figure 6: Block diagram of proposed system initialisation platform for task 1 and task 2

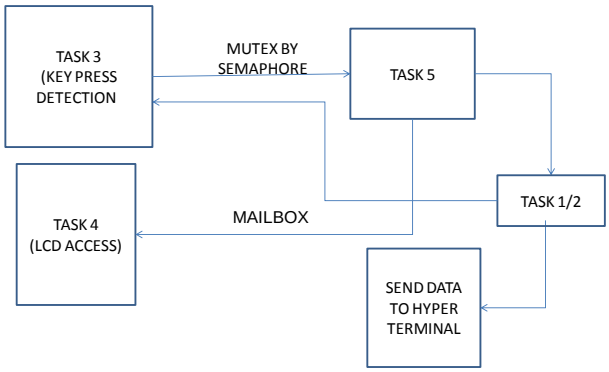

Figure 7: Block diagram of Mailbox, Mutex implementation by use of semaphore

CONCLUSION

In this paper the porting of µC/OS-II in ARM 7 is presented. It mainly focus on designing an embedded system using ARM 7 and µC/OS-II. The steps involved in porting the RTOS and final implementation details are provided.

This paper provides a detailed overview for developing a embedded system using ARM and µC/OS-II and provides

the details about the features of μC/OS-II like Mutex with reference to the Semaphore(which is not the inbuilt one)as well as the inbuilt features like multitasking, scheduling ,mailbox, semaphore.

III. REFERENCES

[1] Liu Zhongyuan, Cui Lili, Ding Hong, "Design of Monitors Based on ARM7and Micro C/OS-II", College of Computer and Information, Shanghai Second Polytechnic University, Shanghai, China, IEEE 2010.
[2] Tianmiao Wang The Design And Development of Em bedded System Based on ARM Micro System and llC/OS-II Real-Time Operating System Tsinghua University Press.
[3] Jean J Labrosse, MicroC/OS-II The Real-Time Kernel, Second Edition Beijing University of Aeronautics and Astronautics Press,