International Journal of Advanced Technology in Engineering and Science www.ijates.com
Volume No.03, Issue No. 03, March 2015 ISSN (online): 2348 – 7550

REAL TIME OPERATING SYSTEM (RTOS) FOR EFFICIENT MULTITASKING USING LPC2148

Prabhakar.B[1], Natarajan.V[2]

[1]PG scholar, Embedded System Technology, SRM University, Chennai (India)
[2]Prof., Embedded system technology, SRM University, Chennai (India)

ABSTRACT

Present scenario is in need of time constraint products with high reliability and productivity. So to provide solution for the problem of time constraint real time operating system is introduced. Real time operating system plays a vital role in time constrained products which meets the deadline requirements of the process. The present work is focused on providing a dedicated real time operating system for the LPC2148 development board which opens up to many application development. It provides a decent platform for application in the name of tasks. Four tasks are taken in which first three tasks run at consecutive intervals and the forth tasks is for monitoring the previous three tasks. All four tasks are expressed by activating the light emitting diode for each task.

Keywords: Free RTOS, Keil, LPC2148, Scheduler, Task.

I INTRODUCTION

The LPC2148 microcontroller is a 16-bit/32-bit ARM7TDMI-S CPU with real-time emulation and embedded tracing support, which combine the microcontroller with embedded high-speed flash memory ranging from 32kB to 512kB. A 128-bit wide memory interfaces with unique accelerator architecture enable 32-bit code execution at the maximum clock rate. LPC means low power consumption and saves energy so it is used in this project.

Most operating systems appear to allow multiple programs to execute at the same time. This is called multi-tasking. In reality, each processor core can only be running a single thread of execution at any given point in time. A part of the operating system called the scheduler is responsible for deciding which program to run when, and provides the illusion of simultaneous execution by rapidly switching between each program. The type of an operating system is defined by how the scheduler decides which program to run when. For example, the scheduler used in a multi user operating system will ensure each user gets a fair amount of the processing time. As another example, the scheduler in a desk top operating system will try and ensure the computer remains responsive to its user.

The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable execution pattern. This is particularly of interest to embedded systems as embedded systems often have real time requirements. A real time requirement is one that specifies that the embedded system must respond to a certain event within a strictly defined time (the *deadline*). A guarantee to meet real time requirement can only be made if the behavior of the operating system's scheduler can be predicted (and is therefore deterministic). Traditional real time schedulers, such as the scheduler used in FreeRTOS, achieve determinism by allowing the user to assign a priority to each thread of execution. The scheduler then uses the priority to know which thread of execution to run next.

II BUILDING BLOCKS

2.1 Development tools

A goal of FreeRTOS is that it is simple and easy to understand. To this end the majority of the RTOS source code is written in C only. Keil 4.22v is used here for compilation and simulation process as well as debugging also.

2.2 RTOS tick

When sleeping, an RTOS task will specify a time after which it requires 'waking'. When blocking, an RTOS task can specify a maximum time it wishes to wait. The FreeRTOS real time kernel measures time using a tick count variable. A timer interrupt (the RTOS tick interrupt) increments the tick count with strict temporal accuracy - allowing the real time kernel to measure time to a resolution of the chosen timer interrupt frequency. Each time the tick count is incremented the real time kernel must check to see if it is now time to unblock or wake a task. It is possible that a task woken or unblocked during the tick ISR will have a priority higher than that of the interrupted task. If this is the case the tick ISR should return to the newly created woken/unblocked task which is effectively interrupting one task but returning to another.

2.3 Signal Attribute

A compare match event on the ARM timer 1 peripheral can be written using the following syntax.

```
void SIG_OUTPUT_COMPARE1A( void ) __attribute__ ( ( signal ) );
void SIG_OUTPUT_COMPARE1A( void )
{
/* ISR C code for RTOS tick. */
vPortYieldFromTick();
}
```

The '__attribute__ ((signal))' directive on the function prototype informs the compiler that the function is an ISR and results in two important changes in the compiler output. The 'signal' attribute ensures that every processor register that gets modified during the ISR is restored to its original value when the ISR exits. This is required as the compiler cannot make any assumptions as to when the interrupt will execute, and therefore cannot optimize which processor registers require saving and which don't. The 'signal' attribute also forces a 'return from interrupt' instruction (RETI) to be used in place of the 'return' instruction (RET) that would otherwise used. The ARM microcontroller disables interrupts upon entering an ISR and the RETI instruction is required to re-enable them on exiting.

2.4 Saving the RTOS Task Context

Each real time task has its own stack memory area so the context can be saved by simply pushing processor registers onto the task stack. Stack memory plays an important role in this work so as the scheduler. Saving the

ARM context is one place where assembly code is unavoidable. So the context is saved in assembly file and converted into a object file that is nothing but a binary file.

2.5 Memory management

The kernel has to dynamically allocate RAM each time a task, queue or semaphore is created. The standard malloc() and free() library functions can be used but can also suffer from one or more of the following problems:

1. Not always available on small embedded systems.
2. Implementation can be relatively large so take up valuable code space.
3. Rarely thread safe.
4. It is not deterministic. The amount of time taken to execute the functions will differ from call to call.
5. It can suffer from memory fragmentation.
6. Can complicate the linker configuration.

Different embedded systems have varying RAM allocation and timing requirements so a single RAM allocation algorithm will only ever be appropriate for a subset of applications. FreeRTOS therefore treats memory allocation as part of the portable layer (as opposed to part of the core code base). This enables individual applications to provide their own specific implementation when appropriate. When the kernel requires RAM, instead of calling malloc() directly it instead calls pvPortMalloc(). When RAM is being freed, instead of calling free() directly the kernel instead calls vPortFree(). pvPortMalloc() has the same prototype as malloc(), and vPortFree() has the same prototype as free().

2.6 Restoring the Context

The RTOS macro port RESTORE_CONTEXT() is the reverse of portSAVE_CONTEXT(). The context of the task being resumed was previously stored in the tasks stack. The real time kernel retrieves the stack pointer for the task then POP's the context back into the correct processor registers.

III PORTING

3.1 Setting up the Directory Structure

The FreeRTOS kernel source code is contained within 3 source files (Co-routine is optional) that are common to all ports, and some 'port' files that contains the RTOS kernel are made for particular architecture.

Steps:

1. Download Free RTOS source code.
2. The files are unzipped into a particular location, to maintain the directory structure.
3. Source code organization and directory structure should be studied clearly.
4. Create a directory that will contain the 'port' files for the particular architecture. Following the convention outlined in the link, the directory structure should be followed properly. For example, if the GCC compiler is used a specific directory for FreeRTOS is created.
5. Copy empty port.c and portmacro.h files into the directory which is created. These files should just contain the stubs of the functions and macro's that require implementing. Existing port.c and portmacro.h files for a

list of such functions and macros. Create a stub file from one of these existing files by simply deleting the function and macro bodies.

6. If the stack on the microcontroller being ported to grows downward from high memory to low memory then set portSTACK_GROWTH in portmacro.h to -1, otherwise set portSTACK_GROWTH to 1.
7. Create a directory that will contain the demo application files for the particular architecture port. Following the convention again this should be of the form created directory and kernel in proper structure, something similar.
8. Copy existing FreeRTOSConfig.h and main.c files into the directory just created. Again these should be edited to be just stub files.
9. Take a look at the FreeRTOSConfig.h file. It contains some macro's that will need setting for chosen hardware.
10. Create a directory off the directory which was created already. Copy into this directory a ParTest.c stub file.
11. ParTest.c consists of 3 simple functions.
12. Set any GPIO pins that can flash a few LEDs,
13. Set or clear a specific LED, and
14. Toggle the state of an LED.

These functions need to be implemented on the development board. Having LED outputs working will facilitate the rest of the required work. The ParTest abbreviation is PARallel port TEST.

3.2 Creating a Project

Now each and every required files are in place that is needed to create a project or makefile that will successfully build them. Obviously they just contain stubs so will not yet do anything, but once they are building the stubs can incrementally be replaced with working functions.

The project will need to contain the following files:

· Source/tasks.c

· Source/Queue.c

· Source/List.c

· Source/portable/Keil/LPC2148/port.c

· Source/portable/MemMang/heap_1.c

· Demo/LPC2148/main.c

· Demo/LPC2148/ParTest/ParTest.c

The following directories need to be in the include path- NOT absolute paths:· Demo/Common

· Demo/LPC2148

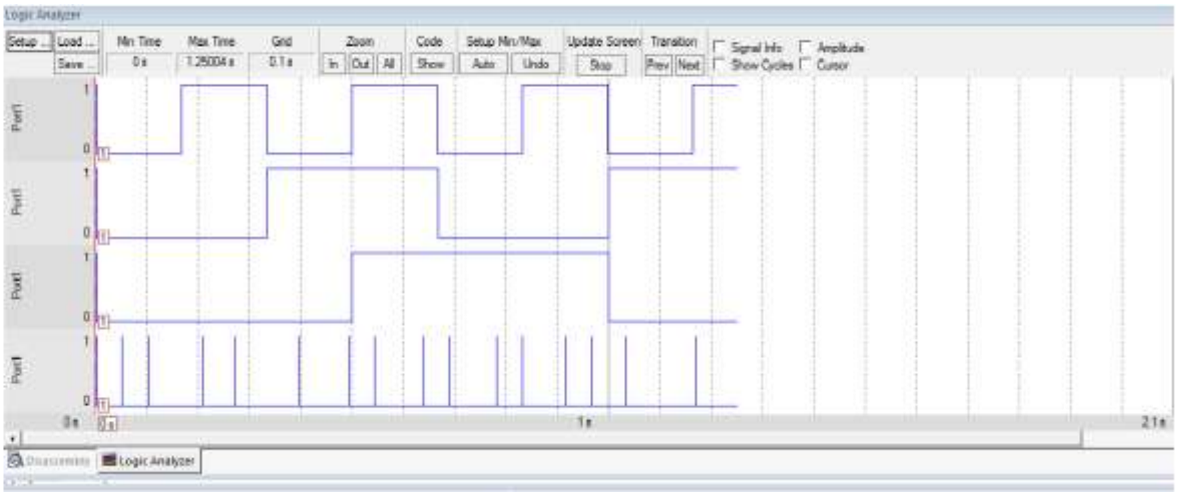
· Source/include

· Source/portable/Keil/LPC2148

3.3 Simulation result

Simulation is done through keil 4.22v (free version). For example that is to show the proper running of

FreeRTOS, four tasks and each task is assigned for each LED which is shown in Fig.1. First task wakes up every 200ms, second task wakes up every 400ms. Third task wakes for every 600ms and fourth task is the supervisor task which should vary from 3ms to 500ms if there is any incomplete task.



Min Time 0 s
Max Time 1.25004 s
Grid 0.1 s

Fig.1: Simulation Result

So the result shows that the tasks are running properly and the scheduler is also doing its job perfectly without entering into the idle mode where the processor gets stopped. If the processor gets stopped means there will no meaning in creating a real time operating system. Now it shows that whatever may be the application, it can be easily controlled by real time operating system.

IV CONCLUSION

The work done on LPC2148 is proved with the simulation result. The compatibility of LPC2148 with freeRTOS is flexible for further improvement on application level. In future using the real time operating system robots can be controlled without any instant control. Once a task is assigned to a robot it finishes and there is no need of manual operation. This is how the paper paves way for the future, as robots are the future of human beings.

REFERENCES

[1] Ching-Han Chen , Sz-Ting Liou , An Embedded Computing Platform for Robot, *SUTC '08. IEEE International Conference*, June 2008.

[2] Su-Lim Tan , Tran Nguyen Bao Anh, Real-time operating system (RTOS) for small (16-bit) microcontroller, *ISCE '09 IEEE 13th International Symposium*, May 2009.

[3] Andrei, Cheng, Efficient Verification and Optimization of Real-Time Logic-Specified Systems, *IEEE Transaction on volume:58, Issue:12,* 2009.

[4] Lortz, Shin, Jinho Kim, MDARTS: a multiprocessor database architecture for hard real-time systems, *Knowledge and Data Engineering, IEEE Transactions on Vol:12, Issue: 4*, 2010.

[5] Richard Barry, *A Practical Guide for using the freeRTOS real time kernel* (2009).