# EMBEDDED REAL-TIME SYSTEMS TO BE APPLIED IN CONTROL SUBSYSTEMS FOR ACCELERATORS

Beibei Shao, Ruopeng Wang, Tsinghua University, Beijing

*Abstract*

Compare the commercial RTOS software package with free software, our choice of system design is primarily based on current application situations in China. The authors have some application experiences of LynxOS, VRTX, pSOS, VXWARK and also some free software, we hope our discussion will give some hints on which system to choose in future control system for the coming several accelerator projects in China. uC/OS and MCX11 is actually a free real-time kernel which has recently been applied in control systems of various areas. And we have found some use of it in accelerator control subsystem because of its exiguity, availability and transparency. The structure of a free RTOS kernel, some basic ideas of real-time system, together with application of the kernels into accelerator control subsystems, is described in the paper.

## INTRODUCTION

Ten years ago, three accelerators were built in China. They are the Beijing Electron Positron Collider, the Accelerator in National Synchrotron Radiation Lab (NSRL) and the Heavy Ion Accelerator in Lanzhou. While in recent a few years Chinese physicists had very difficult time with national financial support. Recently, however, three new accelerator projects are approved. They are the upgrade of NSRL, the electron-cooling ring in Institute of Modern Physics and Shanghai Light Source. Even though, the budget situation for every project will be very tight.

In recent a few years we studied several embedded RTOS. First, there are many free RTOS source code available on the Internet, such as real time Linux, μ/COS, MCX11 and so on. Compared to Institutes of Academia Sinica, there is second cost free resource, which is available only within Universities: companies donate much software for education purpose such as VRTX, pSOS. It makes us a nice condition to learn and evaluate the software and prepare for the three accelerator projects.

The Lynx is a real time Unix. It is not suitable for embedded use.

Although the EPICS is a good choice for low cost solution of making accelerators control system, but it is based on the VXWARK, expensive commercial software. The pSOS, a commercial RTOS package, which has been widely used for communication products development may be used also for accelerator control. There are many library functions for various kinds of device drivers, while the real time kernel itself is a black box for users.

The XRAY from Microtec, which supplies VRTX RTOS, is also evaluated in our lab. The PC windows based program debug environment is a powerful tool for embedded micro-controller application. A program for VME162 module was developed with a remote PC through the Internet in our lab.

In this paper, the usage of free software downloaded from Internet is introduced. The uC/OS has been successfully moved to a MC68332 based micro-controller system. The MCX11 has been used for a data acquisition system in NSRL. It works also in a temperature monitoring system for the slow control of HERA-b experiment in DESY. The advantage of using the free software is:

It is much cheaper than the commercial software.

It has the source code available.

It makes the application program well structured with multi tasks construction and the program is also reliable.

The disadvantage is it needs really a debug tool then the program runs away.

Anyway, just like the free software Linux, which is successfully runs on the PC, the free software source in the Internet can be used also for micro-controller based accelerator sub-system control. Especially in China when the Institute has no enough budget to get commercial RTOS development tools.

## A GLIMPSE INTO REAL-TIME KERNEL

Real-time, multitasking scheme is so complicated that a clear description will fill volumes. The following is depiction of critical nomenclatures in a kernel, which may be helpful to those unfamiliar with such kind of systems.

A real-time kernel provides a software framework within which different processes can operate and gain access to various system resources. Real-time systems usually consist of several processes, or tasks, which need to have control of the system resources at varying times due to the occurrence of external events.

A multitasking real-time kernel promotes an orderly transfer of control from one task to another such that efficient usage of the computer's resources is achieved. Orderly transfers require that the kernel keep track of the needed resources and the execution state of each task so that they can be granted to each task in a timely manner. Response time to a need for kernel services and the execution time of such services must be sufficiently fast enough so that no need goes undetected.

One way to achieve timeliness is the assignment of a priority to each task. The priority of a task is then used to determine its place within the sequence of execution of all tasks. Tasks of low priority may have their execution pre-

empted by a task of higher priority so that the latter can perform some time-critical function.

An event can be any stimulus which requires a reaction from the kernel or a task. Examples of an event would include a timer interrupt, an alarm condition, or a keyboard input. Events may originate externally to the processor or internally from within the software. If response time to any event occurs within a period of time which can be accurately defined and guaranteed, the kernel can be said to be deterministic.

Multitasking appears to give the computer the apparent ability to perform multiple operations concurrently. Obviously, the computer cannot be doing two or more things at once as it is a sequential machine. In multitasking, each task once given operating control either runs to completion, or to a point where it must wait for an event to occur, for a needed resource to become available, or until it is interrupted. Efficient use of the computer can be obtained by using the time a task might otherwise wait for an event to occur to run another task. This switching from one task to another forms the basis of multitasking. The result is the appearance of several tasks being executed simultaneously.

When several tasks can be competing for the resource of execution time, the problem is to determine how to grant it so that each gets access to the system in time to perform its function. The solution is to assign a priority to each task indicative of its relative importance to other tasks in the system. Tasks which have a need to respond rapidly to events are assigned high priorities. Those which perform functions that are not time critical are assigned lower priorities. MCX11 uses a fixed priority scheme in which up to 126 tasks may be defined, while in the kernel uC/OS, tasks have flexible priorities.

The kernel provides an environment whereby two or more tasks can communicate with one another. The three major ways in which this is done are through the mechanisms of semaphore signaling, message transmission, and queues. A semaphore is actually a flag which contains information about the state of the associated event. Any event which is used for task synchronization will be associated with a particular semaphore. The occurrence of a specific event can be signaled by manipulating the semaphore associated with that event.

Message transmission involves the logical transfer of data packets from one task to another. These data packets are called **messages**. Messages are sent from one task and placed in the **mailbox** of the receiving task in the order of the priorities of the senders. Messages may be of any format recognizable by the sender and receiver and data may be passed in either direction. That is, it is possible for two tasks to alternate the roles of sender and receiver.

A third technique whereby two tasks can communicate and synchronize is via a first-in-first-out (FIFO) queuing mechanism. The queuing techniques involve the physical transfer (copying) of data packets from one task to another. Task synchronization due to queuing operations is automatically performed by the corresponding kernels.

Every real-time kernel has a heart beat, which is configured with an interval timer using a real time interrupt clock as a peripheral device. The timer permits task control on a timed basis. A generalized scheme using one-shot and cyclic timers in conjunction with semaphores is provided. The kernel efficiently manages multiple timers using an ordered linked list of pending timer events. A timer for an event is inserted into the linked list in accordance with its duration. Directives for scheduling and canceling timed events are an integral part of the kernel.

Support for a generalized interrupt service scheme is provided within the kernels.

## APPLICATION OF THE KERNELS

The following is applications of the real-time kernels, uC/OS and MCX11, on the micro-controller M68HC11.

uC/OS is a real-time, multitasking, preemptive kernel, which has the following advantages:

- manage up to 63 tasks
- semaphore
- message mailbox and message queue
- priority-based task scheduler, with dynamic task priorities
- clock tick
- can be ported to various micro-controllers, such as those from Motorola, Intel, Hitachi and Zilog

It is quite easy to apply uC/OS in your own system. A typical application includes:

- uC/OS kernel source code and C header file, which incorporates data and routines critical to the whole system.(filename: **UCOS.C, UCOS.H**) This part is irrelevant to the type of micro-controller you applied. So in simple application, no much attention needs to be paid here. You just compile the code together with those closely related to your control target.
- UC/OS hardware-related code, in both C and assembly, which is also a part of the kernel. At the first time you apply uC/OS to a micro-controller, you need to check whether this section is compatible to the hardware and instruction set of your controller. (filename: **UCOS11C.C, UCOS11C.H, UCOS11A.ASM**) For example, during the preservation of context in a interrupt process, the stack frame for different micro-controllers varies a lot in their size, order of registers pushing into or pulling out of the stack. A user has to make sure of this so that your system will not crash when accessing stack.
- Code developed by user. This section is target-oriented code, where tasks stack, priority and entry point are defined, kernel is initialized and then started. So it is the entry point to start the whole system. Just as C programming in DOS environment, there should be a routine named

  **void main()**

  In you code.

The following code list is an example of this section:

```
#include "includes.h"
#define TASK_STK_SIZE 500
UBYTE ExpStk[TASK_STK_SIZE];
UBYTE ExpData;
UBYTE Exp1Stk[TASK_STK_SIZE];
UBYTE Exp2Stk[TASK_STK_SIZE];
UBYTE Exp3Stk[TASK_STK_SIZE];
char string[]="Test string";
void Exp(void *UBYTE);
void Exp1(void *UBYTE);
void Exp2(void *UBYTE);
OS_EVENT *Sema;
void main()
{   OSInit();
   OSTaskCreate(Exp, (void *)ExpData,
     (void*)&ExpStk[TASK_STK_SIZE],10);
   OSTaskCreate(Exp1, (void *)ExpData, (void
     *)&Exp1Stk[TASK_STK_SIZE],20);
   OSTaskCreate(Exp2, (void *)ExpData, (void
     *)&Exp2Stk[TASK_STK_SIZE],1);
   Sema=OSSemCreate(0);
   OSStart();
}
void Exp(void *data)
{   while(1){
      printf("\n\tTask Exp1 Running! ");
      OSTimeDly(60);
 }}
void Exp1(void *data)
{   while(1){
      OSSemPost(Sema);
      printf("\n\tTask Exp2 Running! ");
      OSTimeDly(60);
 }}
void Exp2(void *data)
{  ULONG clk;
   UBYTE err;
   while(1){
      OSSemPend(Sema, 0, &err);
      clk=OSTimeGet();
      printf("\n\t%i ", clk);
 }}
```

MCX11 is a real-time, multitasking, preemptive kernel, which has the following advantages:

- manage up to 128 tasks
- semaphore
- message mailbox and message queue
- priority-based task scheduler, with fixed task priorities
- clock tick
- can only be used on Motorola micro-controller M68HC11

A typical application of MCX11 includes(please compare to the section description of uC/OS):

- MCX11 kernel source code. which incorporates data and routines critical to the whole system.(filename: **MCX.ASM, SYSTEM.ASM**) In simple application, no much attention needs to be paid here. You just compile the code together with those closely related to your control target.
- Code highly related to MCX11 kernel. In this section, the user can define tasks' stack, priority and entry point. And you also need to put the ISR here. Please remember, all the ISR routines should return to a common entry into MCX11 kernel for the system to perform semaphore signaling and task scheduling, if necessary. (filename: **TASKDEF.ASM, ISR.ASM**) Here is an example on how to write code to define tasks:

```
TASK2  equ  STATLS+TCBLEN  *Task 2 TCB addr
STAK2  equ  STKBASE-60      *Addr of task 2 stack
STK2SIZ equ  60              *Size of stack for task 2
     FCB  0                  *INITST is RUN
     FDB initial,STAK2,TASK2
                  *STRTADR,RSTSP,TCBADDR

TASK3   equ   TASK2+TCBLEN  *Task 3 TCB addr
STAK3  equ  STAK2-STK2SIZ  *Addr of task 3 stack
STK3SIZ equ   60             *Stack size for task 3
     FCB   0                 *INITST is RUN
     FDB   read_cmd,STAK3,TASK3
                  *STRTADR,RSTSP,TCBADDR
     …
```

All the tasks are list in the order that their priorities descend. And in a code block for one task, the entry point(such as "initial" and "read"), stack size and TCB(task control block) address is defined or allocated.

- Code to implement the task routines. Normally, every task should be defined as a dead-loop, with waiting on a condition and perform of some functions. Then it is left to the kernel to schedule all these tasks. Finally, a jump to the entry point of the MCX11 kernel initialization will start the whole system.

A environment monitor system has been developed with MCX11. Here 8 tasks as included, which are appointed to maintain devices such as keyboard, displayer, RS-232 communication port, or to perform time-related functions as periodical data acquisition, dose rate calculation. It is much more convenient to have such kernel in your system, because it help promote our software development, and the whole system has less bugs than those developed under the old, unsystematic, single thread scheme.

However, there really lie deficiencies. Since very few debug tools exist for the development, using such kernel requires the user to have deep understanding of real-time system and the specific kernel. This is sometimes not that pleasing. But it is inevitable since we have chosen a free one.

## REFERENCES


[1] Jean J. Labrosse, "uC/OS The Real-Time Kernel" , R&D Publications, August 1993

[2] A.T. Barrett & Associates, "MicroController eXecutive for the Motorola MC68HC11", Version 1.3, Motorola Inc., April 1990