Common Sense and Real Time Executives

William E. Drissel
CyberScribe Associates, Inc., 805 NW 9th, Grand Prairie, Tx 75050

ABSTRACT:
Commercial real-time operating systems are complex and feature-laden. Size and complexity require extensive adaptation and long learning times. For the most common real-life requirements, one can write a real-time executive in one afternoon in any language.

Figures 1 and 2 contrast two views of the world of real time systems - that of the operating system salesman and that of the author.

Commercial products must have every known feature: pre-emptive scheduler, native OS file access, time slicing, message passing, dynamic task creation, priorities, memory allocation, event queues and flags, semaphores, mailboxes, time and calendar functions, etc.

Adaptation to your system is required to prevent the kernel from occupying all available memory. This means sysgening and mastering yards of documentation in which ordinary words are used with extraordinary precision and there are significant penalties for misunderstanding. Contemporary products may have over 50 "system calls" - each requiring register setups and/or a communication block with manifold parameters and options.

Horror stories abound of hundreds of man-hours spent in study of manuals, adaptation and customizing without reaching a stable, useful foundation for development of the real system.

The principal gain we seek from the employment of a real time executive is multi-tasking. I suspect that this term is poorly understood even among programmers. If you have a uniprocessor, its program location counter can be at only one location at any time. Consequently, for some people multi-tasking is a deep mystery. Stripped to its simplest terms, multi-tasking is the ability of a computer program to do more than one thing at a time when observed from outside the computer using human time scale.

Consider, for example, a simple multi-tasking system which polls a telemetry link, prints alarms on a logging printer and updates a wallboard display. When the telemetry link doesn't have input ready, the CPU must be made available to the part of the program which sends characters to the logger. When the printer is busy printing a line, the CPU must be available to update the wallboard. Now, to a human watching the system, all three things seem to be progressing at once even though in a microsecond time scale only one task is occupying the CPU.

If this is the only reason you need a real time executive, you can write your own in one afternoon in any language.

The simplest realtime executive, round-robin, run-til-done:

```
      CALL CLDSTRT
  1   CONTINUE
      CALL ONE
      CALL TWO
      etc.
      GOTO 1
```

In FORTH:

```
: EXECUTIVE
      COLD
      BEGIN
            TASK1 TASK2 TASK3 etc.
      AGAIN
      ;
```

The typical task begins with "why am I here?" case statement:

```
      SUBROUTINE THREE
      GOTO (100,200,300, etc. ), ISTATE3
```

or in Forth:

```
: STATE_MACHINE   ( n task_table —)
      @ SWAP @ CELLS + @EXECUTE
      ;

CREATE TASK1_TABLE
'ACTION1      ,
'ACTION2      , etc.

: TASK1
      STATE1 TASK1_TABLE
      STATE_MACHINE
      ;
```

Typical action tests to see if the world is ready:

```
: ACTIONn
      READY_FOR_THIS
      IF  perform action      THEN
      ;
```

or:

```
100   CONTINUE
      IF (.NOT. READY(WALLBORD)) RETURN
      code to update wallboard
       ...
      RETURN
```

Usually the programmer writing task code knows best when a task is ready to run but some people object to the executive jumping into any code which is not ready to run. This leads us to the idea of a demand scheduler.

```
  1   CONTINUE
      IF  (NEEDED1) CALL ONE
      IF  (NEEDED2) CALL TWO
      etc.
      GOTO 1
```

In FORTH:

```
: EXECUTIVE
      BEGIN
            NEEDED_1
            IF TASK1 THEN
            NEEDED_2
            IF TASK2 THEN
            etc.
    AGAIN ;
```

If a programmer believes priorities are absolutely necessary, the may be accomplished in the following manner:

```
1    CONTINUE
     IF     (NEEDED1)  THEN
            CALL ONE
            GOTO  1
     ENDIF
     IF  (NEEDED2) THEN
            CALL TWO
            GOTO 1
     ENDIF
     etc.
```

In FORTH:

```
: EXECUTIVE
     BEGIN
            NEEDED_1
            IF  TASK1
            ELSE
                  NEEDED_2
                  IF  TASK2 etc.
                  THEN
            THEN
     AGAIN
     ;
```

In 68020 assembler code:

```
EXEC:  BFFFO   NEEDED{0:31},D0   Find first one bit
*              and put bit offset in D0
       JMP     indirect thru table after longword
*              indexing by D0
TABLE: ADDR    TASK1
       ADDR    TASK2
       ....
       ADDR    EXEC    Jump back if no one is ready
```

This last example reaches task code in two instructions! A similar structure can be used to maintain the state machine for a task. If ready to run can be determined by looking at a bit in a command/status register and a conditional branch, the total overhead for cycling through an unready task or reaching action code is six instructions. It would be interesting to compare this complexity with the executive table maintainence and environment restoration code of a commercial real time operating system.

The ideas of priority and demand scheduling combine in a obvious way.

None of the structures above support task suspension (variously called pausing, sleeping, relinquishing). Some people feel that suspension is necessary to allow programmers to maintain a continuity of coding across delays for I/O completion, resource availablity, etc. Since suspension has a considerable cost in complexity, it is worth mentioning that the necessity for suspension can be avoided by task organization. When a task reaches a point where useful continuation isn't possible and control needs to be relinquished to other tasks, the task could simply end. Continuation can be accomplished by another task.

If task suspension is required, we must abandon the simple high-level language approaches shown above. At the moment when a call is made into the suspension code in most modern machines, the stack contains the return address and parent stack frames of the suspending program. These stack frames and the entire environment of the task to be suspended must be preserved while other tasks run and restored before the suspended task is resumed.

The suspension code pushes all the registers and environment (or pointers to environment tables) onto the stack, then saves the stack pointer in an executive queue. To activate the next task, the executive must switch environments by grabbing the next task's stack pointer from one of its queues, restoring registers and environment and issuing a return from subroutine instruction which should resume the task at the instruction beyond the call to the suspension program.

The current fashion of providing even small process control machines with floating point chips and mappers (Motorola 68040 and Intel 80486) greatly increases the amount of environment which must be saved when a task suspends.

Some FORTH systems have been built which use task control blocks containing all the information necessary to resume a task (including USER variables). The first part of the TCB contains a jump instruction to the TCB of the next task when the task is inactive. When the task is ready to run, a subroutine jump, call instruction or interrupt replaces the jump. The call instruction (or alternative) stacks the address of the rest of the TCB and the executive uses this to restore the stack pointer and environment of the task to be resumed. Some machine dependent features can make this very fast and elegant.

To suspend, a task executes one of the executive words provided for this purpose, (typically PAUSE). The executive saves the environment of the calling task in its TCB and then jumps to the TCB of the next task. This chain of jumps eventually finds a call instruction and restores the environment of the task and resumes it.

So far, in the author's practice, suspension has not been found to be worth the trouble.

Since this is a paper about common sense, the next roll-your-own real time executive is mentioned with some hesitation, because its simplicity, elegance and speed might tempt practioners to use it.

This is the interrupt-to-interrupt executive. All tasks except possibly one are connected to interrupts. To switch to a higher priority task, trigger its interrupt. (Incidentally, this provides a suspension mechanism). If you trigger the interrupts of lower priority tasks, they run when the triggering task is finished.

The last sentences contain an important distinction: higher priority tasks seem to run between the trigger instruction and the one which follows; lower priority tasks run after the triggering task completes. Unless the design is perfect the first time, interrupt priorities have to be shuffled and behavior of completed code then becomes unpredictable.

If (nay, when) things get dark and quiet, there are no executive tables to prowl through to determine the active task or which tasks are ready to run because this information is contained in inaccessible flip-flops in the computer. Missing a single interrupt can cause a task (or chain of tasks) to stall. Looking with the debuggers after the fact is no help. All the interrupt catchers are waiting with open arms. There just doesn't seem to be anything for them to do. In spite of its seeming advantages, the development process which involves the interrupt-to-interrupt executive is chaotic.

In this penultimate paragraph, let us dispose of several advertised features which the author perceives to confer little advantage. The first of these is use of mappers to separate tasks. The typical real time system has to *work,* so there is little need for protection from green code — no one plays Star Wars, or develops new code, on the embedded computer which controls the refinery.

The second feature is time-slicing. If time is really pressing, pre-emptive task schedulers and time-slicing take too much time.

The third feature is sometimes called deterministic scheduling. It confers the advantage of being able in some fine-grained sense to say when a task will start. If external reality forces inputs or outputs at specific times, these time marks should cause sychronizing interrupts. The typical computer can respond to such signals in

far less time than an executive can run thru the code to select a task pointer, restore the environment and branch to the task.

This last paragraph contains more common sense in fewer words about the speed of real time systems than anything else I have read. P J Plauger[1] says people worry about the wrong things:

- Those who worry about disk swapping delays should use locked, resident tasks
- Those who worry about task switching times should do more in interrupts
- Those who worry about interrupt latency should use DMA

Reference:

[1]P.J. Plauger. *Evaluating Real Time Operating Systems.* Embedded Systems Programming, February 1990.