Ada Tasking for Sensor and Control Applications

Paul R. Pukite, DAINA pukite@daina.com

Writing control software in the Ada programming language may not rank as a popular method, but given a chance, coding in Ada can pay off in many different ways. In particular, the long-range needs of maintenance and portability can be met through the application of Ada software engineering principles. Among the criticisms levelled against Ada, the debateable issues of too much complexity/size, and low performance lead the way. Opponents also point to the Department of Defense's adoption of Ada in all new applications. The usual argument here is that if the military uses it, it must be obsolete (where "it" is Ada). On the contrary, Ada is one of the few languages engineered from the ground up to support both small and million-lines-of-code applications. In particular, many of the current American and European designs for commercial aircraft flight control systems and air traffic control systems are implemented in Ada.

Assuming a commitment to Ada, a microcomputer application developer will find that the Ada coding constructs allow for development of sophisticated applications. For example, Ada's concurrency features relieves the developer of acquiring a multi-tasking operating system, multi-tasking language libraries, and/or additional code for task scheduling. In fact, the built-in tasking feature of Ada remains a consistently overlooked tool for a PC developer who wishes to design or prototype a multiple sensor-based system. With the recent abundance of relatively inexpensive Ada compilers available for DOS, Windows, and Macintosh platforms, the number of Ada users has steadily increased.

To get a start in Ada, it can't hurt to have a familiarity in Pascal or Modula-2, and to a lesser extent ANSI C or C++ (to get used to stricter type checking). The standardized Ada language reference manual, a robust public-domain Ada interpreter (Ada/Ed) or compiler (GNAT), reusable code libraries, and style guides are available through various sources on the Internet [1] or on an inexpensive CD-ROM [2]. Several books on design and programming practices are also available. For an interesting Ada example, a recent article describes the development of a model railroad controller application for a college class[3].

1. Tasking application

Ada tasking is usually not the first concept introduced during the learning process, but for the sake of differentiating Ada from Pascal or C, it may as well be. In short, including Ada tasks into a piece of code allows an application to contain more than one thread of control. Thus, tasking as used in Ada has the same meaning as in other concurrency driven applications, but is an integral part of the language instead of, say, the operating system. In the following, we design and describe a simple Ada tasking example that allows three parallel tasks to operate.

2. Design

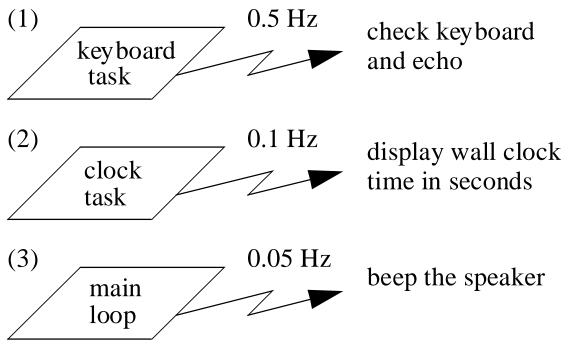
In a generic design for a concurrent application, a high-level process or task will be assigned for each autonomous operation, be it data collection, control function, and so on. Typically, the functions repeat at predetermined time intervals, making a multi-tasking approach a natural mapping to the problem domain. For a PC-based system, some periodic tasking functions that come to mind include: monitoring serial ports or LAN, controlling a printer, background data logging, mouse control, background file backup,

peripheral (DSP, CD-ROM, *etc*.) control, and various dedicated interrupt-to-task mappings.

Our example requires three tasks. They are specified as follows: (1) A task for checking and echoing contents of the keyboard buffer every 2 seconds; (2) A task for displaying the wall clock in seconds at 10 second intervals; (3) A task to beep once every 20 seconds.

We want the tasks to start automatically after loading the program, preferably as transparently as possible. Figure 1 shows the task representation (the parallelograms represent Ada tasks).

FIGURE 1. Task assignments and frequencies.



3. Coding

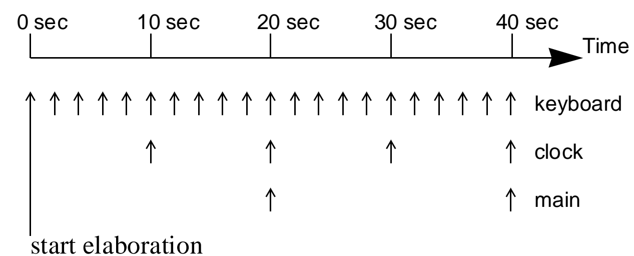
Listing 1 is a non-preemptive Ada program that runs these tasks concurrently. The Ada code is both short and concise. A main procedure called `Task_Demo` encapsulates the three tasks, with the body of the main procedure predefined as a task. The other two tasks, `Keyboard_Handler` and `Seconds_Displayer` are specified and then coded as nested tasks. For each task, the functional code within the loop is assigned to display arbitrary information that would be more complex in a real application (see Table 1 for naming conventions).

To understand the task operation, we focus on the `loop`, `delay`, and `duration` keywords. Once a task starts, it typically enters a `loop`, and then encounters a `delay` statement. At the `delay`, task execution pauses for an amount `duration` (in seconds). During this pause, the Ada run-time searches for other tasks to invoke. If another application task is ready, the run-time will switch execution to that task. Otherwise, the first task's `delay` interval times out and then executes its functional code. This process then repeats via the `loop` keyword.

In effect, `delay` keywords placed within the `loop` bodies allow task switching to take place transparently. Figure 2 shows the mapping of tasks on a timing diagram. The vertical arrows point to where `delay` statements time-out for each task. Note that after every keyboard interval (between the arrows), the Ada run-time checks for other running tasks, which may or may not be scheduled to occur at multiples of the keyboard interval. In this way, Ada task scheduling is not based on *absolute* time but rather on *relative* times. Thus, scheduling should take into account the delay plus the task execution time. The coder must ensure that sufficient time for switching between the tasks occurs, otherwise the specified delay intervals may need to be modified (At the lower limit, a delay of 0.0 seconds would simply check for any other tasks of equal or higher priority before returning).

The example listed here demonstrates non-preemptive tasking; for preemptive Ada tasking, a waiting task will preempt or interrupt a running task if it is ready to run and has a higher priority. For the most part, if task functions are a fraction of the cycle period, an application consisting of many tasks can be automatically scheduled using the non-preemptive approach. Preemption or mapping interrupts to task entry points should be used for application tasks with higher urgency or stricter timing deadlines.

FIGURE 2. A timing diagram for task scheduling assuming that task execution is instantaneous.



4. Compiling

The tasking example application was compiled with the Meridian Ada version 4.1 DOS compiler [4]. The entire process requires four steps. (1) We set up a new directory to hold a program library (for example, `c:\ada\demo`). Then at the DOS prompt within the new directory, issue the commands (2) `newlib`, (3) `ada taskdemo.ada`, and (4) `bamp`, to create a directory, compile the code, and then build (i.e. link) the Ada main program.

If changes are required in the code, then we only have to rebuild the code with the Meridian Ada utility `amake.exe`. This is possible as the compiler automatically maintains the library information up to date, without the programmer needing to write a special makefile.

5. Running the example

The following is typical output, where the 'a' key is pressed, a pause occurs for 1/2 a minute, and then the 'b' key is pressed.

```
a->a Time: 24525.8700
 Time: 24535.9100 (beep)
 Time: 24545.9800
 Time: 24556.0300 (beep)
b->b Time: 24566.0699
 Time: 24576.1200 (beep)
 Time: 24586.1700
^C
```

Stopping the program requires a control break. Notice that the time between clock updates is not exactly 10 seconds because the display code executes in a fraction of a second (this can be taken care of by adjusting the `duration` according to the system clock).

This is a simple example. Most Ada environments can also access assembler, in-line machine code, make OS calls, map to ports or interrupts, link C-routines, and represent or pack data. Most operating systems are supported, including Microsoft Windows. More complex applications for controlling or monitoring instrumentation are well within the reach of Ada's capabilities.

References

[1] File transfer protocol (FTP) sites are constantly changing, therefore use an Internet tool such as Archie, Veronica, or Gopher to do a search for FTP directories or files matching Ada, Ada-Ed, GNAT (GNU Ada Translator), etc. (case sensitivity may be important).

[2] Walnut Creek CD-ROM, Suite 260, 1547 Palos Verdes Mall, Walnut Creek, CA 94596, 1(510) 674-0783.

[3] J.W. McCormick, A Model Railroad for Ada and Software Engineering, Communications of the ACM, November 1992, p.68 (special section on Ada).

[4] Meridian, 10 Pasteur St., Irvine, CA 92718, 1(800) 221-2522. Now part of Rational Software, Inc.

TABLE 1. Like Pascal, Ada is not type-sensitive, so we use capitalization to improve readability. Lower-case terms signify Ada reserved keywords. Terms like `CHARACTER` and `ASCII.BEL` are system dependent but universally defined. Library packages such as `Calendar` and `Text_IO` are standard and portable to most environments, while `TTY` and `SPIO` are provided with the specific Ada (Meridian) compiler.

Identifier	Case	Examples
Built-in reserved	Lower	`task, delay, begin, with, package`
System library	Upper	`CHARACTER, ASCII.BEL`
Standard Ada library	Mixed	`Text_IO, Calendar`
Other libraries	Mixed	`TTY, SPIO`
User-defined	Mixed	`Time, Char, Keyboard_Interval, Task_Demo`

Listing 1 : Complete code for `task_demo.ada`

```
with Calendar;  -- library package for system clock
with Text_IO;   -- library package for file and standard IO
with TTY;       -- library package for console IO
with SPIO;      -- library package for special IO functions

procedure Task_Demo is

   package Clock_IO is new Text_IO.Fixed_IO( duration );

   Keyboard_Interval : constant duration := 2.0;  -- in seconds
   Clock_Interval    : constant duration := 10.0;
   Beep_Interval     : constant duration := 20.0;

   task Keyboard_Handler;  -- Specification
   task Seconds_Displayer; -- Specification

   task body Keyboard_Handler is
      Char : CHARACTER;
   begin
      loop  -- Echo keyboard at frequency 1/Keyboard_Interval
         delay Keyboard_Interval;
         while TTY.Char_Ready loop
            Char := TTY.Get;
            Text_IO.Put( "->" );
            Text_IO.Put( Char );
            SPIO.Flush;
         end loop;
      end loop;
   end Keyboard_Handler;

   task body Seconds_Displayer is
      Time : duration;
   begin
      loop  -- Display clock at frequency 1/Clock_Interval
         delay Clock_Interval;
         Time := Calendar.Seconds( Calendar.Clock );
         Text_IO.Put( " Time:" );
         Clock_IO.Put( Time );
         Text_IO.New_Line;
      end loop;
   end Seconds_Displayer;

begin
   loop     -- Beep speaker at frequency 1/Beep_Interval
      delay Beep_Interval;
      TTY.Put( ASCII.BEL );
   end loop;
end Task_Demo;
```