A Turbo Pascal Task Scheduler

1. Introduction

This Turbo Pascal unit allows the programmer to do a crude form of scheduling within a Turbo Pascal program. This is written in Turbo Pascal 4.0 and should do alright at 5.0. This unit can be used however you want, and distributed freely, so long as there is no charge for it. I also make no claims as to efficiency or bugs and will not be considered liable for use, misuse or bugs in the code. So if you build an important application around this and something goes wrong, don't expect to sue me, or expect that much help for that matter. If you are writing applications for public domain or yourself, then contact me through the BIBMUG BBS in Buffalo, New York and perhaps I'll give you a hand. That's not meant to be nasty, I just don't want to be beholden to anyone over this little bit of code.

First off, what does it do. This Unit allows you to schedule procedures do be executed. These are executed according to specific scheduling criteria established for each procedure and for the program as a whole. The way this all works is that you

1. load the procedures with individual scheduling criteria
   into the procedure stack using add_task.
2. set the system criteria (optional)
3. determine the current schedule point
4. execute the procedures that meet the system and current
   scheduling criteria using run_tasks.

Sounds worse then it is.

2. System Criteria.

There are four types of system scheduling criteria. These are define by the enum *task_schedule_criteria*. This is defined below :

```
task_schedule_criteria = (
        task_criteria_mod,          {default.}
        task_criteria_equal,
        task_criteria_more,
        task_criteria_less);
```

The first criteria is *task_criteria_mod* for modulus. In this a procedure is executed if (current criteria MOD task criteria) equals 0. This is useful for scheduling procedures to run periodically. For example, assume proc_1 has a task scheduling criteria of 10 and it is in code like this :

```
for i := 1 to 100 do
        run_tasks(i);
```

Then proc_1 will be run 10 times, every 10 cycles of the loop. This would also be useful in clock driven systems where every X clock ticks you execute a procedure.

The next criteria is *task_criteria_equal* for equals. While the system is in this mode, procedures whose task schedule number matches the current schedule number will be executed. For example, say you have four procedures proc1, proc2 proc3 and proc4 with scheduling criteria of 1,2,3 and 4 respectively. Now in this loop :

```
for i := 1 to 10 do
        for j := 1 to 4 do
                run_tasks(j);
```

each proc would be run once every cycle for a total of 10 times each in succession. Another example is this :

```
for i := 1 to 10 do begin
        if odd(i)
                then j := 1
                else j := 2;
        run_tasks(j);
        end;
```

where proc1 and proc2 alternate being executed, each running 5 times.

The last two criteria *task_criteria_more* and *task_criteria_less* are used to set threshold conditions. Basically when these are set a procedure is executed every time the current scheduling criteria is greater then or less then the task's scheduling criteria. In the next example assume the system scheduling criteria is task_criteria_less and there is a procedure called fix_stuff in the stack whose task schedule criteria is 5 (make five fixes):

```
fixes := 1;
while true do begin   {go until doomsday}
        do_something;
        run_tasks(fixes);
        if fixed then   fixes := fixes + 1;
        end;
```

As you can see the procedure fix_stuff will be run until five (5) fixes have been made (whatever that means). Under optimal conditions one would assume that means five times. Now if you expand this to include a proc called bigfixes with a criteria of 10 then you'd get 5 fixes with 10 bigger fixes. The task_criteria_more option works the same way in the opposite direction. So in the above example, fix_stuff would not be executed until five fixes were detected and bigfixes until 10 fixes were detected. This is also useful for running a program that is clock driven but aperiodic. One could set it so a program only executed after so many clock ticks had already transpired.

3. The Procedure Stack and Procedure Scheduling.

Procedures are entered on the procedure stack using the procedure add_tasks. The address of the procedure to be run and the scheduling criteria (a longint number) are passed to it and a task number is returned. The function itself returns an error code. A code of task_ok means it was successfully added to the stack. Programs that meet the current and system criteria are run in the order they appear on the stack. Thus the stack only contains pointers to the procedures to be run and their scheduling criteria and doesn't take up much space in memory. There is a limit of 100 procedures that can be placed on the stack. Attempting to add more would result in an error of task_full from add_task. The task number is used for later reference. The number of tasks is a constant *task_limit* that is currently set to 100.

All procedures placed on the stack must be of the same form. That is **procedure <procedure name>(schedule : longint);** . Thus the current scheduling criteria is the only information passed to the procedure, that is the scheduling criteria passed to run_tasks. Any deviation will cause a problem. Also any procedure that is to be placed on the stack must be compiled with the {$F+} compiler option (force far calls). This allows the program to search for the procedure outside the units' code segment. This is important. If the F parameter isn't on, then the program will lock up and will freeze up you PC.

4. Error Messages

A variety of error messages are available. They are constants that return the following

```
task_ok          =   0;
task_full        =   1;
task_empty   =   2;
task_illegal     =   3;
task_none        =   4;
```

The status task_ok shows that everything is alright. task_full is returned by add_tasks to indicate that the stack is empty. task empty is returned by run_tasks to show that there are no tasks to run. task_illegal is issued in response to an illegal task number being used (if the number is < 1 or greater the task_limit) and task_none is used as a result of attempting to perform an operation on a task that hasn't been loaded into the stack such as delete_task or change_schedule.

5. The Functions.and Procedures.

These are the functions and procedures that make up the tasks unit.

function add_task(schedule : longint; member : pointer;
task_number : byte) : byte;

Add_task places a procedure on the stack and sets up its scheduling. It returns a task number used for reference later and an error code (as part of the function call). the

scheduling criteria is a longint number and the reference to the procedure is made by passing the address of the procedure to add_tasks. Add_tasks returns either task_ok or task_full if the stack is full. For example :

```
error := add_tasks(10, @proc1, task_num);
where proc 1 is defined as procedure proc1(schedule :longint);
if error = task_full
        then writeln('Stack filled up');
```

The task number is used to refer to it later.

function add_task_number(task_number : byte;schedule : longint;
member : pointer) : byte;

Add_task_number places a procedure on the stack in the same way that add_task does, but in a specific place. This is used to fill in the "holes" left by deletes. It cannot be used to add a task to the end of the stack, only add_task does that. The space for the intended addition must be unoccupied (previously deleted) and legal. If it is occupied or in any way illegal, including being the end of the stack a task_illegal message is returned.

function space_left : byte;

This tell you how much stack space is left for add_tasks. This does not taking into account deleted tasks.

function first_space : byte;

This returns the place of the first open space or "hole". If there are no open spaces then zero (0) is returned. If there are no deleted spaces then the end of the stack is returned.

function delete_task(task_number : longint) : byte;

This deletes a task on the stack. The memory that it occupied is released, and its space is free for another task. To fill it in later use add_task_number, not add_task. Add task only adds to the end of the stack. It returns task_illegal if you attempt to delete a task that isn't there or is out of range.

function change_schedule(task_number : byte;
schedule : longint) : byte;

This function changes the scheduling criteria of the task task_number. As in the other functions, task_illegal is returned if a bad task number is passed to it. It replaces the old criteria with the new one.

procedure set_criteria(task_criteria : task_schedule_criteria);

Set_criteria set the system criteria which is of type task_schedule_criteria. See System Criteria (chapter 2) for more information.

function run_tasks(schedule : longint) : byte;

This function run any procedure on the stack whose individual criteria meet the current criteria in relation to the system criteria. For more information on scheduling see above. This returns task_empty if there are no tasks to run.

function run_task_number(task_number : byte) : byte;

This function executes task number <task_number> immediately regardless of criteria. I'm not sure what this could be used for but, what the heck. It automatically passes the value zero to your procedure.

Index