

## **A Turbo Pascal Task Scheduler**

1. Introduction 2
2. System Criteria. 2
3. The Procedure Stack and Procedure Scheduling. 4
4. Error Messages 4
5. The Functions.and Procedures. 4

## 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 to 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.

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.

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

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`.

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

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

"hole"5  
{F+} compiler option4  
Add\_task4  
Add\_task\_number5  
Aperiodic3  
BIBMUG BBS2  
Change\_schedule5  
Delete\_task5  
error messages  
    error messages4  
First\_space5  
Force far calls4  
Functions4  
Procedure stack4  
Procedures4  
Run periodically2  
Run\_task\_number6  
Run\_tasks4, 6  
Set\_criteria6  
Space\_left5  
System scheduling criteria2  
Task\_criteria\_equal3  
Task\_criteria\_less3  
Task\_criteria\_mod2  
Task\_criteria\_more3  
Task\_empty4  
Task\_full4  
Task\_illegal4  
Task\_limit4  
Task\_none4  
Task\_ok4  
Task\_schedule\_criteria2  
Threshold conditions3