The Task Manager
Version 2.2.1

The Task Manager is a package for creating and managing tasks—separate execution threads that run nonpreemptively in the background. Tasks should periodically call a Task Manager yielding routine to allow other tasks to run. Tasks are ideal for lengthy processes that you would like to run in the background, since the task runs in a separate execution thread from your event loop.

The Task Manager was written using THINK C 5.0.1, and requires the MacTraps library. If you are using THINK C 4.0, see the section below called "Using the Task Manager With THINK C 4.0."

This package consists of the following files:

Task Manager Folder:	contains entire package
TaskTest.π	project for sample application
TaskTest.c	source code for sample application
Task Manager Notes	this documentation (Word 4.0)
Task Manager Notes.txt	this documentation in text–only form
Task Manager Source:	folder containing the source code
Task.c	Task Manager source code
Task.h	Task Manager include file

Overview

The Event Loop

A simplified Macintosh event loop looks something like this:

```
void EventLoop( void )
{
        Boolean         gotEvent;
        RgnHandle               cursorRgn;


        cursorRgn = NewRgn();
        do {
                gotEvent = WaitNextEvent( everyEvent, &theEvent,
                GetSleep(), cursorRgn );
                AdjustCursor( theEvent.where, cursorRgn );
                if( gotEvent )
                        DoEvent( &theEvent );
                else
                        DoIdle();
```

```
		} while( TRUE );
	}
```

The application calls `WaitNextEvent` to retrieve the next event from the event queue. The application–defined routine `GetSleep` determines the minimum number of ticks to allow the operating system to wait before returning control to this application. When a null event occurs, the application calls its `DoIdle` routine to perform background operations. This is when the Task Manager should be called to run background tasks:

```
	void DoIdle( void )
	{
		RunTasks( GetWake());
	}
```

At idle time, the application calls the Task Manager routine `RunTasks` to process background tasks. The single parameter to `RunTasks` is the minimum number of ticks that background tasks may use. This time is calculated by the application–defined routine `GetWake`.

The Task

The task itself is an application–defined routine that may look something like this:

```
	void MyTaskProc( long taskRefCon )
	{
		Boolean		moreToDo;


		moreToDo = TRUE;
		while( moreToDo ) {

			/* Allow other tasks to run */
			TaskYield();

			/* Do my task processing */
			…
		}
	}
```

The task procedure should periodically call the Task Manager routine `TaskYield`, which allows the Task Manager to run other tasks. When `TaskYield` is called, the Task Manager checks whether the number of ticks specified in the call to `RunTasks` has been exceeded, or if there are any events pending for the application. If there is no reason to suspend the task, then `TaskYield` simply returns. Otherwise, the task is suspended and control returns to `DoIdle`, immediately after its call to `RunTasks`. The application can then cycle through its event loop once more. The next time the application calls `RunTasks`, control will return to `MyTaskProc`, immediately following the call to `TaskYield`.

This flow of control may seem a bit crazy at first. There are two loops—the event loop and the loop within the task procedure—which appear to be executing *simultaneously*. This is the beauty of the Task Manager. It allows you to write your tasks as simple loops (nested loops if necessary), and merely call an idling routine to relinquish control periodically. Without the Task Manager, either you would need to “unroll” the loops in your task proce dures and turn them into finite state machines, or you would need a second, more restrictive event loop that you could call from within your task procedure.

The Task Manager accomplishes this magic by maintaining separate *execution threads* for each task, and by *switching context* between threads when running different tasks. This is very similar to the way that the Finder will switch contexts between applications. Task switching within the Task Manager, however, is much more efficient, since all tasks are still part of the same application, and therefore share low–memory globals and all other application–specific data structures.

Task Term Procedures

For each task, you can define a special procedure that will be called when the task is disposed of. This can happen

- when the task returns from its task procedure (it finishes its task normally),
- when any task calls `DisposeTask` for it (the user closes a window associated with the task, or presses *command–period*),
- or when the application calls `TermTasking` (the user quits the application).

Sleep and Wake Times

The application spends its time alternating between periods of sleep and wake. It does its work while awake, and other applications do their work while it is asleep. You must choose a good balance of sleep and wake times so that your application can get its work done without too much expense from other applications.

There are two factors that should affect your application's sleep–wake cycle: whether the application is in the foreground or background, and whether the application has work to do (tasks) at the moment. This table shows the four resulting situations and gives sample values for wake and sleep (all values are in ticks):

	in Foreground	**in Background**
has Tasks	high wake (60) low sleep (0)	low wake (30) medium sleep (30)
no Tasks	high wake (60) high sleep (12000)	low wake (30) high sleep (12000)

There are a few things to note here:

- When there are no tasks, we sleep for a ridiculously long time. The operating system will wake us when we receive an event. Otherwise, there's nothing to do, so we let other applications run.

- However, there's no need to adjust our wake time based on number of tasks, since `RunTasks` will return immediately when there are no tasks.

- We presume that the user is focused on the foreground application, and wants it to get more time than applications in the background. When we are in the foreground and busy, we work for one second, then poll for events and allow other applica tions to work briefly. When we are in the background, we spend less time awake and more time asleep.

The Stack

The key to creating separate execution threads is to create a separate stack for each one. The stack location itself doesn't change, but when a task is suspended, the current stack contents are copied into a Handle. When the task is resumed, the contents of this Handle are copied back to the stack.

While a task is running, its stack handle is made purgeable. When the task yields, this handle is reallocated to the current stack size. The stack handle is allocated from temporary memory, if possible. Otherwise, it is allocated from the application's heap.

This method is quite reliable. And, since the amount of stack space that a task will use is typically quite small, it proves to be fairly efficient as well.

Task Manager Routines

Task Manager Initialization

Before using the Task Manager routines, your application should call `InitTasking` to set up the Task Manager's data structures:

```
OSErr InitTasking( void );
```

If the Task Manager could not be initialized due to memory limitations, `InitTasking` will return the appropriate system error code.

Creating Tasks

To create a new task, call `NewTask`:

```
OSErr NewTask( TaskProcPtr taskProc, TaskProcPtr
    taskTermProc, long taskRefCon, short
    *taskRefNum );
```

where

`taskProc`	is the task procedure
`taskTermProc`	is the task term procedure. You may pass a NULL pointer if you don't need a term procedure.

`taskRefCon`	is a parameter that is passed to the task
`taskRefNum`	is the task's reference number. You may pass a NULL pointer if you don't need the reference number.

`NewTask` returns a Memory Manager error code if there is not enough memory available for the task's data structures or stack. The task's stack will be allocated from temporary memory if it is available.

A `TaskProcPtr` is defined as:

```
    typedef void ( *TaskProcPtr )( long taskRefCon );
```

The task procedure and the task term procedure can either be static or extern C functions, but their CODE segments should remain loaded until the task procedure returns. If the task term procedure is in a different segment from the task procedure, then the task term procedure can unload the task procedure's segment.

Disposing of Tasks

When a task has finished its work, its task procedure simply returns. It will be automatically removed. To dispose of a task *before* its task procedure returns (for example, if the user closes a window associated with a task, clicks "Stop," or presses *command–period*), call `DisposeTask`:

```
    OSErr DisposeTask( short taskRefNum );
```

where

`taskRefNum`	is the task's reference number.

`DisposeTask` returns an error code (`paramErr`) if the task reference number is invalid.

`DisposeTask` calls the task's term procedure, if one is defined.

Task Manager Termination

When you have finished using the Task Manager, or when your application exits, call `TermTasking`:

```
    OSErr TermTasking( void );
```

This routine calls `DisposeTask` for all running tasks and restores any traps that were patched by `InitTasking`. It is extremely important to call `TermTasking`, since if you don't, important traps won't be restored and your machine may crash when it exits.

`TermTasking` must be called from the context of your application. If called from a task procedure, it returns `paramErr`.

Letting Tasks Run

Your application should call `RunTasks` in response to a null event or a `FALSE` return code from `GetNextEvent` or `WaitNextEvent`:

```
void RunTasks( unsigned long wakeTime );
```

where

`wakeTime`	is the amount of time you want allocated to background tasks.

The constant `kDefaultWakeTime` is a reasonable choice for the `wakeTime` parameter. However, you should keep track of whether your application is in the foreground or the background and whether tasks are running, and use this information to determine a more suitable wake time. See the discussion of sleep and wake times above, for more information.

Occasionally, your task procedure should call `TaskYield` to allow other applications and other tasks within your application to run.

```
void TaskYield( void );
```

`TaskYield` will return control to the application (that is, the application will return from it's call to `RunTasks`) if there are no more tasks to run, if the wake time specified in `RunTasks` has elapsed, or if there is an event pending for the application.

Keeping Track of Tasks

Your application can keep track of its tasks by calling these Task Manager routines:

```
short CountTasks( void );
```

`CountTasks` returns the number of tasks that the Task Manager is currently running.

```
short CurrentTask( void );
```

`CurrentTask` returns the task reference number of the current task, or 0 if there is no current task (if called from the application, for example).

```
short GetIndTask( short index );
```

where

`index`	is the number of the nth task counting from 0 to `CountTasks()` – 1

`GetIndTask` returns the task reference number of the given task, or 0 if `index` is invalid.

```
long GetTaskRefCon( short taskRefNum );
```

where

`taskRefNum`	is the reference number of the task whose reference constant you want

`GetTaskRefCon` returns the reference constant for the given task, or 0 if `taskRefNum` is invalid. This is useful if you want to send messages to your tasks by changing parameters that are pointed to by the tasks' reference constants.

```
OSErr SetTaskRefCon( short taskRefNum, long
    taskRefCon );
```

where

`taskRefNum`	is the reference number of the task whose reference constant you want to set
`taskRefCon`	is the new value for the task's reference constant

`SetTaskRefCon` changes the reference constant for the given task. This is useful if you want to send messages to your tasks by modifying their actual reference constants.

`SetTaskRefCon` returns `paramErr` if `taskRefNum` is invalid.

Debugging Tasks

The Task Manager contains some debugging checks designed to uncover common errors. To enable these checks, simply edit "Task.h" and change the macro `TASK_DEBUG` to 1.

When TASK_DEBUG is set to 1, the Task Manager checks for the following error conditions:

- **Stack overflow**—This can occur when the heap grows beyond a suspended task's stack and restoring its stack would cause the heap to be overwritten.

- **WindowRecord in stack**—Due to the way the stack is managed, it is not a good idea to allocate nonrelocatable parameter blocks in the stack (i.e., as local variables). A common example of this occurs when allocating a WindowRecord on the stack. To help track down this case, the Task Manager will check the WindowList for WindowRecords that were allocated in the stack.

- **`RunTasks` called from task**—`RunTasks` should only be called from the application.

- **`TaskYield` called from application**—`TaskYield` should only be called from a task procedure.

- **Unable to save environment**—This can occur if a task's environment can't be saved due to memory constraints.

Using the Task Manager With THINK C 4.0

The Task Manager will work just fine under THINK C 4.0. It requires the ANSI and MacTraps libraries, and the Gestalt function. You need the MacTraps library from Symantec which contains the Gestalt glue. If this is not available, contact me and I will supply you with temporary glue you can use.

ANSI

The Task Manager uses the `setjmp` and `longjmp` routines from the THINK C ANSI library. If you don’t want to include ANSI, then include the file “setjmp.c” in your project.

With THINK C 5.0, the ANSI library is not needed.

Profiler

Under THINK C 4.0, “Task.c” will not work if it is compiled with the Profiler code generation option. To circumvent this, create a project (“Task.π”) containing only “Task.c.” Turn the Profiler code generation option off and compile “Task.c.” In your application’s project, remove Task.c and replace it with Task.π.

You should also avoid profiling across `TaskYield` calls. Set the global variable `_profile` to `FALSE` before calling `TaskYield`, then set it back to `TRUE` afterwards.

Under THINK C 5.0, the code is immune to any option settings.

Limitations

Task Procedures

You are only limited by available memory to the number of tasks you can create and run simultaneously.

Task procedures can call any Toolbox routine (including QuickDraw). However, the current grafPort is not preserved across calls to `TaskYield`. If you have several tasks drawing to different ports, or if you draw in your application code, you may need to call `SetPort` after calling `TaskYield`.

Local Variables

Since the stack contents are not guaranteed to remain constant while `TaskYield` is executing, you should not allocate parameter blocks as local variables. For example, the following code will not work if `pb` is a pointer to a local variable:

```
OSErr TPBOpen( ParmBlkPtr pb )
{
        OSErr           err;


        /* Clear ioCompletion */
        pb->ioCompletion = 0;

        /* Open the file asynchronously */
        err = PBOpen( pb, TRUE );
        if( err != noErr )
                return err;

        /* While waiting for the open to complete… */
        while( pb->ioResult == 1 ) {
                /* …allow other tasks to run */
                TaskYield();
        }

        /* Completed; return the result */
        return pb->ioResult;
}
```

To make this routine function properly, your parameter block should be a global or static variable, or allocated from the heap.

Version History

Version 1.0

- Initial Release.

Version 1.1

- Add `TermTasking` routine to circumvent crashes when an application exits under Finder.

Version 1.2

- Fix bug in `GetTaskRefCon`.
- Use `StripAddress` on the address passed to `NSetTrapAddress`, to avoid any nasty 32-bit addressing problems.

Version 2.0

- Make the Task Manager work with THINK C 5.0. It will compile correctly if the “Require Prototypes” and “Check Pointer Types” are enabled. It will also compile correctly with a <MacHeaders> that was built with “Check Pointer Types” turned on.
- Rename `TaskIdle` to `TaskYield`, which is more descriptive. For compatibility, a macro is included that maps `TaskIdle` to `TaskYield`.
- By popular demand, add task term procedures.

Version 2.2

- Rework the way the Task Manager handles the stack. See the section “The Stack” above for more details.

Version 2.2.1

- Fix a bug in the `SaveEnvironment` routine that caused task switching to crash. This routine requires certain parameters and variables to live in registers.

Coming Attractions

Please send me mail regarding *any* new features you would like to see added to this package.

Yours Truly

I welcome any comments or suggestions that will help me improve or extend the functionality of the Task Manager. You can reach me at:

Internet:	Michael_Hecht@mac.sas.com
AppleLink:	SAS.HECHT

Happy Tasking!
— Michael Hecht