

CoExec: A Simple Task Switcher for Dynamic C

SHDesigns, Version 1.2

S. Henion

July 2, 2006

<http://shdesigns.org/rabbit/coexec-dc.html>

Table of Contents

1.0 GENERAL DESCRIPTION.....	3
2.0 COMPARISON TO DC EXEC.....	4
3.0 EXEC INTERFACE.....	4
3.1 Exec Initialization.....	5
3.2 Creating Tasks.....	6
3.3 Task Switching.....	7
3.4 Starting and Stopping Tasks.....	8
3.5 Aborting and Restarting Tasks.....	8
3.6 Events.....	8
3.7 Semaphores.....	9
3.8 Stack Checking.....	10
4.0 PORTING FROM THE DC MODEL.....	10
4.1 Porting a Simple Program.....	10
4.2 Porting More Complex programs.....	12
4.3 Stacks.....	12
5.0 DEBUGGING.....	13
6.0 PREEMPTIVE TASK SWITCHING.....	14
7.0 EXAMPLE APPLICATION.....	15

1.0 GENERAL DESCRIPTION

The simple task switcher is a cooperative task manager. It is designed for easy migration from the DC cofunc and costate model. It does not emulate this model, this operates differently than the DC task switcher. This exec is referred to as CoExec.

The DC syntax is non-standard. There is no simple way to emulate the syntax with an ANSI compiler. Also, the DC method is inefficient, it involves copying data on/off the stack to do task switching. This exec provides similar functionality. Some of the DC syntax is emulated for ease of porting.

The DC costate/cofunction model has become too unreliable in the recent compilers. CoExec was written originally for Softools to help port DC code.

This exec provides simple cooperative task switching along with some minimal preemptive switching. If you need more advanced features, you should use something like uCos.

Features added in 1.2:

1. Event support added.
2. Fixed need for empty tasks[] entry.
3. Fast task switching, about 6us @ 44mHz.
4. Removed need for alignment on stacks, done automatically

Features added in 1.1:

1. Supports far stacks, this saves root memory space and allows each task to have.
2. Allows main stack to be split.
3. Faster task switching <50us on a 22mHz R2000
4. Preemptive support now more robust.
5. Task switch counters removed.
6. Function task_swap removed. Now task_switch can be called directly from a timer function.
7. restart_task now needs param to pass to task (param no longer stored in globals)

CoExec is licensed to a single developer. There are no per-product or per-unit charges.

2.0 COMPARISON TO DC EXEC

The DC task switcher is cooperative and its operation is hidden in global structures. For simple tasks, this works fine. Nested costates and recursive cofunc functions are complex and hard to debug. Also, nested costates has proven to be buggy in many versions of the DC compiler.

SHDesigns developed CoExec to help port DC programs to Softools C. Later development in DC found serious problems in Dynamic C's costates. They were unreliable and no work-around was found. The project just used about 8 costates in a main loop. Even this simple app would fail. So, CoExec was ported to DC and has been used successfully.

The CoExec operates differently. This may use more memory as each task has a different stack; however, memory usage can be tuned per task. Having individual stacks has a definite advantage. Each tasks auto variables are isolated. This allows recursion to operate without corrupting the task structure like the DC model. The DC model requires memory for all costates and cofunctions regardless if they are run. It also needs to copy data on and off the current stack to switch costates and call cofunctions.

Tasks run until they reach a command that asks for a task switch. Then the next task stack pointer is loaded and the new task runs. There is a provision to allow preemptive task switching. This only handles the case where one task does not do a task switch in a reasonable time. It also provides some of the slice functionality of DC

One of the main differences is there is no such thing as a cofunc. Any function can be called from any task. In the DC model, if a function calls a cofunc, it must also be defined as a cofunc. This makes programming much easier. If you needed to add a call to a cofunc to an existing function, you needed to change the current function to a cofunc and any functions that call it would need a waitfordone call. This makes simple code changes possibly complex.

In DC, the stack is 4096 bytes. That means all tasks share this same 4k. CoExec allows each task to have its own 4k stack. This stack is allocated from xmem and takes no root space.

CoExec also supports primitive preemptive support. This will prevent one task from hogging the CPU for long periods. It operates similar to the “slice” mode of DC.

3.0 EXEC INTERFACE

The exec is added to an application via using the “#use co_exec.lib” statement. Be sure to add the full path to co_exec.lib in the LIB.DIR file in the DC installation directory.

The exec only has a few functions define in co_exec.lib:

```
// initialization
int init_tasks(int num_t);
int create_task(int (* func)(int),void * stack,unsigned stack_size,int param);

// task switching
void sleep(unsigned long stime);
void task_switch(void);
```

```

// task start/stop
void task_run(int tasknum,int run);

// task abort/restart
void task_abort(int tasknum);
int restart_task(int tasknum,int param);
int check_abort(void);
int check_exit(int task_num);

// stack checking
int check_stack(int tasknum);

//events
// wait for an event to be set
// returns state of event (1 if set, 0 if max_time reached)
int wait_event(char * eventflag,unsigned long max_time);

// set event and put tasks waiting in run state
// returns number of tasks waiting
int set_event(char * eventflag);

// clear an event
void clr_event(char * eventflag);

```

There are three global variables used:
extern unsigned char task_count; // 0 until first task created
extern TASK tasks[];
extern TASK * _cur_task; // current task data

These are described in the following sections.

3.1 Exec Initialization

The CoExec lib uses a TASK array to describe tasks. This must be defined in the user program. An example is as follows:

```

#include co_exec.lib

#define NUM_TASKS 6
TASK tasks[NUM_TASKS];

```

NUM_TASKS defines the number of tasks. This constant must be used in the init_tasks() function so the exec knows the size of the array. So in the start of main() call the init_tasks() function:

```

int init_tasks(NUM_TASKS);

```

This function initializes the task structures to known values. It also set up the current task (main()) as task #0. Task 0 will always be the main() loop.

When creating threads with xmem stacks, the create_task() function will try to align stacks so they do not cross a 4k boundary. This relies on xalloc() working in a particular way. With DC 9.25, memory is allocated downward. In other versions of DC or with different compiler options, this may not be the case. Test two xallocs() in a test program. If the second has a lower address than the first, it will work. If the second has a higher address, define the following before the #use co_exec.lib:

```
#define NEGATIVE_XALLOC 0
```

That will tell CoExec that xmem is allocated in ascending order.

Creating stacks that intentionally hit 4k boundaries will reduce wasted xmem. I.e. Allocating 512, 1024,2048,512 will cause the next to be on a 4k boundary.

3.2 Creating Tasks

Task 0 is the main() thread. It is created automatically.

Tasks are created with the create_task call:

```
int create_task(int (* func)(int),void * stack,unsigned stack_size,int param);
int create_named_task(int (* func)(int),char * name,void * stack,unsigned stack_size,int param);
```

Each task should have the following prototype:

```
int task(int param)
{
}
```

The parameters to create_task are as follows:

func	- This is the address of the function that will be run
stack	- a char[] array that will be used for the stack
stack_size	- the size of the stack
param	- this integer will be passed to the function

The create_named_task() also has a “name” parameter. This pointer is stored in the task structure so it can be later be recalled by get_task_name(int tasknum).

Once a task is created it will run on the next requested task switch. Normally each task has a while () loop that does not exit.

The return value from create_task() is the task number. This number must be used in task control calls that need a task number. If the task can't be created (exceeded NUM_TASKS or no xmem for far

stacks.)

The param can be any integer value. This is not used by the exec. The param is useful for creating multiple threads with the same function. A different param value will allow each thread to have its own "id". This param could be used as an index into a shared array.

If stack==NULL, the stack is allocated from the far memory pool allocated by xalloc(). If a static buffer address is passed, it will be used instead of a far stack. Note: tasks created using near stacks can only be created from the main task.

Local variables in tasks can not be accessed by other tasks if a xmem stack is used. This is because the stack is not in the memory map when other tasks run. Any task with local (auto) variables that need to be passed to other tasks must use a static, global stack (char stack[] passed to create.)

Be sure to follow the stack sizes as described in init_tasks(). The Rabbit can only map memory on 4k boundaries. So stacks must be less than or equal to 4096.

3.3 Task Switching

A task switch is initiated by a "yield" statement. "yield" is #defined as task_switch() in co_exec.lib. It also matches the DC yield functionality.

If a task is waiting for some device to be ready, it could call yield, i.e.:

```
int ser_task(int param)
{
    while (!check_abort())
    {
        if (buffer_full())
            yield;
        /// do processing here
    }
    return 0;
}
```

The above task waits for the buffer_full() function to return 0. If non-0 it yields to another task.

Using yield in a loop may not be very efficient. Usually a task would want to wait a amount of time before checking if it can continue. You would be tempted to use delayMS() to wait, but the function does not allow other tasks to run while it is waiting for the delay time.

The sleep() function implements this in an efficient way. In DC delayMS() had to be repeatedly called to wait for a delay time. In CoExec, this delay is implemented in the exec itself. When sleep() is called, the task run flag is cleared. The time to wake up the task is calculated. When this time is passed, the run flag is set and the task will run on the next task switch.

3.4 Starting and Stopping Tasks

A task can be stopped and started. Stopping a task simply marks the task as non-run able and it will be skipped on a task switch. The function to control tasks is as follows:

```
void task_run(int tasknum,int run)
```

Where:

tasknum is the task ID returned from create_task()

run =0 to stop, run=1 to continue

An alternative is to have a task only loop when an event is set and sleep on it cleared. The event calls allow this (see 3.6)

3.5 Aborting and Restarting Tasks

A task can be aborted. This does not actually stop the task. It simply flags the task as "aborted." The task itself handles the abort. An abort-able task would operate as follows:

```
int near task1(int param)
{
    while (!check_abort()) // do until we're told to abort
    {
        // task loop here
    }
    return -1;    // return value could indicate an error
}
```

The check_abort() will return the abort flag. When this is set, the task should exit. The task_abort() function is used to signal a task to exit.

When a task is created, its exit value will be set to 0. The check_exit() function can be used to check if a task has exited. Any task that exits should use a non-0 return value to allow the check_exit() function to be used.

A task can be restarted. The restart_task() function will reinitialize a task to start up just the same as it was called from create_task(). Note, a task should never restart itself! Also, the main() task should never be restarted.

Any task can exit on its own without an task_abort() call. It will then set the exit status. The check_exit() function will return 0 if the task is still running. A task should return a non-0 value, otherwise other tasks would have no way to tell if the task has exited.

3.6 Events

The exec has basic event handling. The code does not protect the events from multiple access. An event is just a char memory flag. A task waits on an event by entering sleep() with a pointer to an event set in the task. If the event is set, the sleep() is cleared and the task is set to the run state.

```
int wait_event(char * eventflag,unsigned long max_time);
```

The wait_event() call will wait for an event. The max_time is the time to wait. There is no infinite wait implemented. The state of the event is returned. If the event is not set and the max_time delay is reached, the function will return 0.

```
int set_event(char * eventflag);
```

The set_event() call sets an event, signals any waiting tasks. This will switch to any waiting task on the next task switch..

```
void clr_event(char * eventflag);
```

This simply sets *eventflag=0;

3.7 Semaphores

CoExec does not implement semaphores directly. It does include some macros to implement semaphores.

Semaphores are needed for shared variables (or devices). They may need to be protected by the SEM_LOCK() and SEM_UNLOCK() macros. An example of their use follows:

```
char _disk_io_sema=0; // semaphore to protect the disk controller
```

```
void disk_lock(void)
{
    while (1)
        if (SEM_LOCK(_disk_io_sema)==1)
            return;
        else
        {
            SEM_UNLOCK(_disk_io_sema);
            sleep(1); // a longer value may be better here to prevent excessive task switching.
                    // if maximum response speed is needed, use yield;
        }
}
```

SEM_LOCK(variable) will return the # of locks on a semaphore. IF >1 then the task must call SEM_UNLOCK() and try again.

```
void disk_unlock(void)
{
    SEM_UNLOCK(_disk_io_sema);
}
```

3.8 Stack Checking

When a task is created, its stack is filled with 0xff's. This allows the stack area to be checked to see how much memory has been used. The stack is used by function calls, auto variables and interrupts. If the stack overflows, the system will become instable.

An excessively large stack for each task would waste memory. Sometimes the amount of stack needed can not be easily calculated. The stack check function allows this to be determined at run-time.

The `check_stack(int task_num)` function returns the number of bytes untouched in the task stack. This is done by searching the stack from the bottom and finding the first byte that is not 0xff.

4.0 PORTING FROM THE DC MODEL

Porting code from the DC method of task switching can be simple; but complicated costate and cofunc DC programs will be difficult.

4.1 Porting a Simple Program

The simplest to port are programs that have just a set of costates in the `main()` loop. The following example would be easy to convert:

```
cofunc int my_cofunc(int param)
{
}

main()
{
// init code omitted
    while (1)
    {
        costate
        {
            wfd my_cofunc(0);
            wfd my_cofunc(1);
            yield;
        }
        costate
        {
            waitfor(delayMS(100));
            flash_led();
        }
    }
}
```

The first step is to make the costates separate functions. One of the costates will remain in the main loop. Any others will have to be made separate tasks. In the example, the second costate is made a function called `led_task()`:

```

int led_task(int param)
{
    while(!check_abort())
    {
        sleep(100);
        flash_led();
    }
    return 1; /// will never get here but gets rid of compiler warning
}

```

The `waitfor(delayMS())` call has been replaced with a `sleep()` call. It can be used as is; `waitfor()` is defined as follows:

```
#define waitfor(x) while (!(x)) sleep(WAITFOR_DELAY)
```

`WAITFOR_DELAY` defaults to 0, so it is effectively a `yield`; statement. If `WAITFOR_DELAY` is `#defined` before the lib is included (`#use Co_exec.lib`), it will sleep for the specified time instead of just `yield'ing`.

This will help reduce the “task churning” of the DC model. The `DelayMS()` function can be called thousands of times, just to wait a few milliseconds. This causes excessive task switching. Usually, the `waitfor()` test only needs to be done every few milliseconds. Defining `WAITFOR_DELAY` to 5 will still give tasks 5ms response to the `waitfor()` and reduce unneeded tests. `WAITFOR_DELAY` defaults to 0 to emulate the DC model more accurately.

We will have 2 tasks, `main()` and `led_task()`. We will need to create them. Main is already there, so the only task needed will be the `led_task()`.

```

// add include for co_exec defines
#include co_exec.lib

#define NUM_TASKS 2
#define COEXEC_XMEM 4096
TASK tasks[2];

char led_stack[512]; // we need a stack for the led task

main()
{
// init code omitted
    // init the exec
    init_tasks(NUM_TASKS,COEXEC_XMEM);
    create_task(led_task,NULL,4096,1);
    while (1)
    {
        wfd my_cofunc(0);

```

```

        wfd my_cofunc(1);
        yield;
    }
}

```

The above code will run the two tasks. Note: the `co_exec.lib` will add defines to remove any `cofunc`, `scofunc`, `wfd` and `waitfordone` statements. These are no longer needed and can be removed for clarity.

At the start of the source file, add a `"#use co_exec.lib"` to add the `CoExec` prototypes and defines. The `co_exec.lib` should be added to the project `DC LIB.DIR` file.

4.2 Porting More Complex programs

The porting example in 4.1 is fairly straightforward. This section describes how to port other DC functions.

The cofunctions do not really change. Indexed cofunctions may not be needed. The index was needed to allow separate `costate` variables to be used in one function. In most cases the index can be simply removed. Separate "state" information is maintained by using separate stacks for each task.

The `waitfor()` function can be emulated. It can be replaced with:

```

waitfor(some_statement);

while (some_statement==0)
    yield;

```

The `co_exec.lib` defines a macro to make this change automatic. For performance, it is usually better to replace the `yield` with a `sleep()` time. This helps reduce unneeded task switching.

Netsted `costates` will be difficult to port. These are not a good idea anyway as it gets complicated and was buggy in the DC compiler.

Named `costates` can be emulated with named tasks. The `exec` commands for `run`, `stop`, `abort` and `restart` a task can be used to emulate the `costate` commands to `start/stop/rerun` a named `costate`.

4.3 Stacks

Each thread will need a stack. The size of the stack must be sufficient for each task.

A task switch uses about 14 bytes of the stack. Any interrupt will use the stack of the current task, so all tasks must have sufficient stack to handle interrupts and task switches. A minimal stack for a task that makes little function calls could be as small as 100 bytes.

Each task that calls functions will need to have stack space for the function call and its local (auto) variables and parameters.

Testing has shown that a stack size of 512 bytes is plenty for most tasks. Usually 256 bytes is

sufficient. However, any task that calls the TCP/IP stack may need a large stack (up to 3k if DHCP is used.) The main() task would normally have a large stack and should be used for most TCP/IP calls.

Functions with a lot of local variables will need a larger stack. If these functions are not called by more than one task, they can use static local variables to save stack space.

When using xmem stacks, each task can have up to 4096 bytes in its own stack. This uses no root data. This allows much more auto variables than regular DC cofunctions/costates. There is one issue. Data in the stack can not be shared with other tasks. Also, sockets can not be auto on a task stack. Other tasks will call tcp_tick() and they will not have access to the other task stack when they call tcp_tick() and a packet comes in for another socket in another rthread. Declare all sockets "static" to get around this issue.

TCP or UDP sockets can not be shared. So if you create two identical threads, you will need to have then use separate static sockets. The parameter passed to the task can be used for this. It can be an index into an array of socket structures or cast to a pointer to the structure.

5.0 DEBUGGING

If COEXEC_VERBOSE is defined, the exec will output printf() messages on task creation, exit, restart or abort.

The only issue debugging is setting a breakpoint in a function that is called by more than one task. There is no way to specify what task the breakpoint applies to, so the breakpoint will stop when any task calls the function.

6.0 PREEMPTIVE TASK SWITCHING

CoExec allows preemptive task switching. The `task_switch()` function is designed to operate from an interrupt. `task_switch()` will switch to the next available task.

The `task_switch()` can be called from a timer interrupt() like from the timer B library:

```
TimerBInit(500000l,1); // do 20 task switches/sec
```

The include a timer B routine to do the switch:

```
root void TimerBRoutine(void)
{
#ifdef DEBUG_RST
    if (dkStatusFlags&0x4) // do task switch if running
#endif
    yield;
}
#endif
```

The check for `dkStatusFlags` is needed to prevent other tasks from running while stopped in the debugger.

Preemptive task switching should be avoided unless it is really needed. It works, but adds possible problems with shared variables.

When using preemptive task switching, the user must consider if the functions that the tasks call are reentrant. Also, shared variables may not always be valid. If one task is in the middle of modifying a global structure and a task switch occurs, the next task would read the partially modified data.

The interrupt must save all registers before calling `task_switch()`. The `task_switch` will save only IX and IY. It should also clear the source of the interrupt to prevent an immediate call from the next task.

7.0 EXAMPLE APPLICATION

The example application is `coop_task.c`. It has several tasks:

`kbd_task()` - reads chars from the stdio window and echoes them.

`fast_task()` - this task does nothing but yield; in a loop.

`slow_task()` - this task never yields, requires preemptive task switching for other tasks to run. If `timerB` is used it will never yield. If not used, it will only yield once every few seconds. This is an example of a non-cooperative task.

`task()` - simple task that `sleeps()` for a time specified by the param passed to it. Three copies of this task will be run.

The `main()` task starts the tasks and reports the task switch counts to the `printf()` output. It also stops and restarts the `slow_task()`. Once every 10 seconds it reports the amount of stack free in each task.

Every 20 seconds the `slow_task` is aborted then restarted 10 seconds later. This demonstrates the ability to abort and restart a task. It also demonstrates preemptive tasking. The `slow_task` never gives up the CPU. It is not cooperative. While running the demo, look at the counts for `fast_task()`. When `slow_task` is hogging the CPU, the fast task is only run about 4 times a second. Since `slow_task` is not cooperating, the other tasks can only run when a task switch interrupt occurs. When `slow_task` is aborted, `fast_task` switches many times a second. This would be the normal way cooperative multi-tasking should work. The slow task is included to show the limitation and how preemptive tasking can help when non-cooperative tasks are used.

If `USE_TIMERB` is #defined, preemptive task-switching is enabled. This will require the `timerb.lib` available from: <http://shdesigns.org/rabbit/freelibs.html> See section 6.0 on precautions using preemptive task switching.

Version 1.2 adds an event task. This task is signaled from the main loop once every 15 seconds. The task just waits for the event to be set, then clears it.

The demo uses both a global stack and allocates stacks from `xmem` stacks. It will report the free stack area of each task once every 20 seconds.