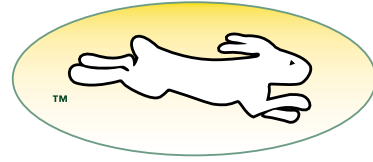


Application Note



AN205

How to Get a μ C/OS-II Application Running

Introduction

μ C/OS-II is a highly configurable, real-time operating system. It can be customized using as many or as few of the operating system's features as needed. This application note outlines:

- The configuration constants used in μ C/OS-II,
- How to override the default configuration supplied in `UCOS2.LIB`.
- The necessary steps to get an application running.

It is assumed that the reader has a familiarity with μ C/OS-II or has a μ C/OS-II reference ([MicroC/OS-II, The Real Time Kernel](#) by Jean J. Labrosse is highly recommended).

Default Configuration

μ C/OS-II usually relies on the include file `os_cfg.h` to get values for the configuration constants. Since Dynamic C does not use this header file, these constants, along with their default values, are in `UCOS2.LIB`. A default stack configuration is also supplied in `UCOS2.LIB`. μ C/OS-II for the Rabbit uses a more intelligent stack allocation scheme than other μ C/OS-II implementations to take better advantage of unused memory.

The default configuration allows up to 10 normally created application tasks running at 64 ticks per second. Each task has a 512-byte stack. There are 2 queues specified, and 10 events. An event is a queue, mailbox or semaphore. You can define any combination of these three for a total of 10. If you want more than 2 queues, however, you must change the default value of `OS_MAX_QS`.

Some of the default configuration constants are listed below.

```
#define OS_MAX_EVENTS          10 // Max number of events (semaphores,
                                // queues, mailboxes)
#define OS_MAX_TASKS          10 // Max number of tasks (less stat
                                // and idle tasks)
#define OS_MAX_QS              2 // Max number of queues in system
#define OS_MAX_MEM_PART       0 // Max number of memory partitions
#define OS_TASK_CREATE_EN     1 // Enable normal task creation
#define OS_TASK_CREATE_EXT_EN 0 // Disable extended task creation
#define OS_TASK_DEL_EN        0 // Disable task deletion
#define OS_TASK_STAT_EN       0 // Disable statistics task creation
#define OS_Q_EN                1 // Enable queue usage
#define OS_MEM_EN              0 // Disable memory manager
#define OS_MBOX_EN            1 // Enable mailboxes
#define OS_SEM_EN              1 // Enable semaphores
#define OS_TICKS_PER_SEC      64 // number of ticks in one second
#define STACK_CNT_256         1 // number of 256 byte stacks (idle
                                // task stack)
#define STACK_CNT_512 OS_MAX_TASKS + 1 // number of 512 byte stacks (task
                                // stacks + initial program stack)
```

If a particular portion of $\mu\text{C}/\text{OS-II}$ is disabled, the code for that portion will not be compiled, making the overall size of the operating system smaller. Take advantage of this feature by customizing $\mu\text{C}/\text{OS-II}$ based on the needs of each application.

Custom Configuration

In order to customize $\mu\text{C}/\text{OS-II}$ by enabling and disabling components of the operating system, simply redefine the configuration constants as necessary for the application.

```
#define OS_MAX_EVENTS          2
#define OS_MAX_TASKS          20
#define OS_MAX_QS              0
#define OS_MAX_MEM_PART       15
#define OS_TASK_STAT_EN       1
#define OS_Q_EN                0
#define OS_MEM_EN              1
#define OS_MBOX_EN            0
#define OS_TICKS_PER_SEC      64
```

If a custom stack configuration is needed also, define the necessary macros for the counts of the different stack sizes needed by the application.

```
#define STACK_CNT_256         1 // idle task stack
#define STACK_CNT_512         2 // initial program + stat task stack
#define STACK_CNT_1K          10 // task stacks
#define STACK_CNT_2K          10 // number of 2K stacks
```

Follow the $\mu\text{C}/\text{OS-II}$ and stack configuration constants with a **#use "ucos2.lib"** statement. This ensures that the definitions supplied outside of the library are used, rather than the defaults in the library.

```
#use ucos2.lib
```

This configuration uses 20 tasks, two semaphores, up to 15 memory partitions that the memory manager will control, and makes use of the statistics task. Note that the configuration constants for task creation, task deletion, and semaphores are not defined as the library defaults will suffice. Also, note that 10 of the application tasks will each have a 1024 byte stack, 10 will each have a 2048-byte stack, and an extra stack is declared for the statistics task.

Examples

The following sample programs demonstrate the use of the default configuration supplied in `ucos2.lib` and also a custom configuration that overrides the default.

Example 1

In this application, ten tasks are created and one semaphore is created. Each task pends on the semaphore, gets a random number, posts to the semaphore, displays its random number, and finally delays itself for three seconds.

Looking at the code for this short application, there are several things to note. First, since $\mu\text{C}/\text{OS-II}$ and slice statements are mutually exclusive (both rely on the periodic interrupt for a “heartbeat”), `#use ucos2.lib` must be included in every $\mu\text{C}/\text{OS-II}$ application (1). In order for each of the tasks to have access to the random number generator semaphore, it is declared as a global variable (2). In most cases, all mailboxes, queues, and semaphores will be declared with global scope. Next, `OSInit` must be called before any other $\mu\text{C}/\text{OS-II}$ function to ensure that the operating system is properly initialized (3). Before $\mu\text{C}/\text{OS-II}$ can begin running, at least one application task must be created. In this application, all tasks are created before the operating system begins running (4). It is perfectly acceptable for tasks to create other tasks. Next, the semaphore each task uses is created (5). Once all of the initialization is done, `OSStart` is called to start $\mu\text{C}/\text{OS-II}$ running (6). In the code that each of the tasks run, it is important to note the variable declarations. The default storage class in Dynamic C is static, so to ensure that the task code is reentrant, all are declared auto (7). Each task runs as an infinite loop, and once this application is started, $\mu\text{C}/\text{OS-II}$ will run indefinitely.

```

#use ucos2.lib // 1. Explicitly use uC/OS-II library
void RandomNumberTask(void *pdata);
OS_EVENT* RandomSem; // 2. Declare semaphore global so all tasks have access
void main() {
    int i;
    OSinit(); // 3. Initialize OS internals
    for(i = 0; i < OS_MAX_TASKS; i++) {
        OSTaskCreate(RandomNumberTask, NULL, 512, i);
        // 4. Create each of the system tasks

        RandomSem = OSSemCreate(1);
        // 5. Create semaphore to control access to random-number generator

        OSStart(); // 6. Begin multitasking
    }
}

void RandomNumberTask(void *pdata) {
    auto OS_TCB data; // 7. Declare as auto to ensure reentrance
    auto INT8U err;
    auto INT16U Random;

    OSTaskQuery(OS_PRIO_SELF, &data);
    while(1) {
        // Rand is not reentrant, so access must be controlled
        // via a semaphore.
        OSSemPend(RandomSem, 0, &err);
        Random = (int)(rand() * 100);
        OSSemPost(RandomSem);
        printf("Task%d's random number: %d\n", data.OSTCBPrio, Random);

        // Wait for 3 seconds in order to view output from each task.
        OSTimeDlySec(3);
    }
}

```

Example 2

This application runs exactly the same code as Example 1, except that each of the tasks are created with 1024-byte stacks. The main difference between the two is the configuration of μ C/OS-II.

First, each configuration constant that differs from the library default is defined. The configuration in this example differs from the default in that it allows only two events (the minimum needed when using only one semaphore), 20 tasks, no queues, no mailboxes, and the system tick rate is set to 32 ticks per second (1). Next, since this application uses tasks with 1024 byte stacks, it is necessary to define the configuration constants differently than the library default (2). Notice that one 512 byte stack is declared. Every Dynamic C program starts with an initial stack, and defining `STACK_CNT_512` is crucial to ensure that the application has a stack to use during initialization and before multi-tasking begins. Finally `ucos2.lib` is explicitly used (3). This ensures that the definitions in (1 and 2) are used rather than the library defaults. The last step in initialization is to set the number of ticks per second via `OSSetTicksPerSec` (4).

The rest of this application is identical to Example 1.

```
// 1. Define each of the necessary configuration constants for uC/OS-II
#define OS_MAX_EVENTS      2
#define OS_MAX_TASKS      20
#define OS_MAX_QS         0
#define OS_Q_EN           0
#define OS_MBOX_EN        0
#define OS_TICKS_PER_SEC  32

// 2. Define each of the necessary stack configuration constants
#define STACK_CNT_512      1           // initial program stack
#define STACK_CNT_1K      OS_MAX_TASKS // task stacks

// 3. This ensures that the above definitions are used
#include ucos2.lib

void RandomNumberTask(void *pdata);

// Declare semaphore global so all tasks have access
OS_EVENT* RandomSem;

void main() {
    int i;

    // Initialize OS internals
    OSInit();
    for(i = 0; i < OS_MAX_TASKS; i++) {
        // Create each of the system tasks
        OSTaskCreate(RandomNumberTask, NULL, 1024, i);
    }
}
```

```

// Create semaphore to control access to random-number generator
RandomSem = OSemCreate(1);

// 4. Set number of system ticks per second
OSSetTicksPerSec(OS_TICKS_PER_SEC);

// Begin multitasking
OSStart();
}

void RandomNumberTask(void *pdata) {
// Declare as auto to ensure reentrance
auto OS_TCB data;
auto INT8U err;
auto INT16U Random;
OSTaskQuery(OS_PRIO_SELF, &data);
while(1) {
// Rand is not reentrant, so access must be controlled
// via a semaphore.
OSSemPend(RandomSem, 0, &err);
Random = (int)(rand() * 100);
OSSemPost(RandomSem);
printf("Task%02d's random number: %d\n", data.OSTCBPrio, Random);
// Wait for 3 seconds in order to view output from each task
OSTimeDlySec(3);
}
}

```



Rabbit Semiconductor
2932 Spafford Street
Davis, California 95616-6800
USA

Tel. (530)757-8400
FAX (530)757-8402

Web site: <http://www.rabbitsemiconductor.com>