Introduction to the MicroC/OS RTOS for the NetBurner MOD54415

What is MicroC/OS ?

MicroC/OS is a real-time operating system (RTOS), also called a “kernel”, for microcomputers, which has the following features:

- *pre-emptive multitasking* and *unique priorities per task*
- many typical *kernel services* including task management, time delays, semaphores, mutual exclusion semaphores, event flags, mailboxes, message queues, etc.
- *portability*, because it is written in standard ANSI C
- *scalability*, because kernel features can easily be included or left out using conditional compilation
- *determinism*, which means that the response times for most kernel services are predictable and do not depend on the number of tasks that are currently active

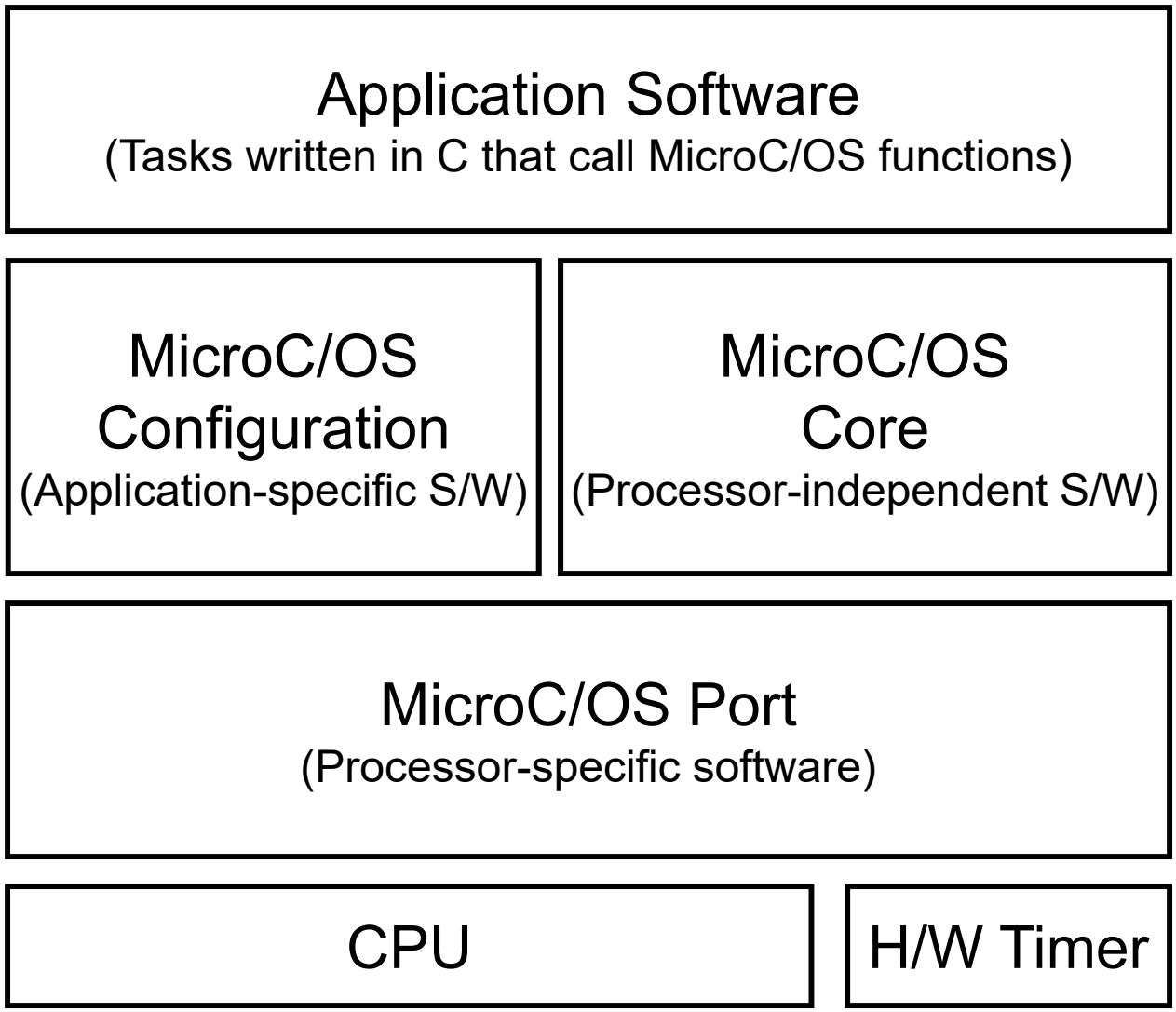
A Brief History of MicroC/OS

- Designed by Jean J. Labrosse in the early 1990s.
- The first version, called μCOS or MicroC/OS, was released into the public domain in the summer of 1992.
- A new version was released along with a book titled *μC/OS, The Real-Time Kernel*, (CMP Books, 1992).
- Enhanced version released along with a book titled *MicroC/OS-II, The Real-Time Kernel*, 2nd ed., (CMP Books, 2002).
- MicroC/OS-II and -III are distributed by Micrium Inc. MicroC/OS-II can be used royalty-free by educational institutions for teaching & research. *Note*: A license is required to include the kernels in commercial products.
- The NetBurner boards that we use in ECE 315 use a modified version of the original royalty-free MicroC/OS.

MicroC/OS Variations

- Free “ports” (that it, software implementations that have been modified to work on a different computer) of MicroC/OS are available for a wide variety of different CPUs/microcomputers and IDEs/compilers.
- The version of MicroC/OS that we have for the NetBurner MOD54415 lab microcomputer is slightly different from the version described in the 2002 book by Jean Labrosse.
- The process of editing source C/C++ code, compiling it, linking together a system “build”, downloading the build to flash memory on the MOD54415, and then executing the new build has been automated within the Eclipse IDE. Such automation increases productivity.
- The function call sequence for initializing the MicroC/OS environment has been simplified, and some MicroC/OS function calls have been dropped and new ones added.

System Architecture Using MicroC/OS



The MicroC/OS Core

- The core of MicroC/OS provides services that can be invoked by user programs by calling functions in C.
- User software is assumed to be partitioned into 1 to 63 tasks, which are written in C. Each task is usually structured in the form of an infinite loop that blocks on system calls.
- Each task is associated with:
  1. A "task priority", from 1 to 63, where 1 is the highest priority and 63 is the lowest priority

     63 (the lowest priority) is reserved for the **IdleTask**.

     38, 39, 40, 45, 50, 51, 52 & 63 are reserved in our NetBurner implementation of MicroC/OS. *User tasks should only use priorities in the range 53 to 62*.
  2. Program code that defines the task behaviour.
  3. A private stack region in memory.
  4. An optional private task-specific data structure, which can store a name, floating-point register contents, etc.

MicroC/OS Core (cont'd)

- MicroC/OS and MicroC/OS-II always create a special **IdleTask**, which is run on the CPU when there is no other task that is on the ready-to-run queue.
- MicroC/OS-II optionally creates a StatisticsTask, which is capable of gathering useful system performance data.
- The NetBurner version of MicroC/OS does not have the StatisticsTask, but similar capabilities are provided by the OS functions OSDumpTasks and OSDumpTCPStacks.
- **At any given time, the task with the lowest priority that is also ready-to-run will be using the CPU.** If no user task is currently ready-to-run, then either the IdleTask (or the StatisticsTask, if present) will be running on the CPU.
- In MicroC/OS, the first user task "UserMain" (1) initializes the kernel, (2) initializes the TCP/IP stack, (3) starts the HTTP server, (4) creates an initial set of user tasks, and (5) carries on executing, usually at a low frequency. UserMain might be used later to shut down the system.

Reserved NetBurner MicroC/OS Priorities

- The NetBurner version of a networked MicroC/OS environment reserves priority 63 for the IdleTask, and 0 for the highest priority task (reserved and not available).
- The following additional priorities are also reserved:

```
#define MAIN_PRIO (50)  // for the UserMain task
#define HTTP_PRIO (45)  // for web server task
#define TCP_PRIO  (40)  // for TCP layer task
#define IP_PRIO   (39)  // for IP layer task
#define ETHER_SEND_PRIO (38)  // for UDP sends
```

- User tasks must use other priorities. In the lab system, priority 51 is used by the LED task, and 52 controls the seven-segment display. *Priorities 53 to 62 are available*.
- See C:\Nburn\include\constants.h for priority definitions.

Application Software for MicroC/OS

- The application software gains access to MicroC/OS by including specific global include files.
- It is useful to define symbols for various constant values so that meaningful names, not obscure numbers, can be used later.
- The hidden main() procedure loads up the context switch TRAP vector, initializes MicroC/OS system data structures, creates the default **UserMain()** start-up task, and starts multitasking.
- Code before the UserMain task allocates storage for the stacks of any other tasks, and space for any other "global" data structures (e.g., variables, pointers, semaphores, message queues, etc.) that will be required. These objects will be accessible by all tasks. Code must be provided for all of the user tasks, which are not executing at first.
- The UserMain task finishes **system initialization**, finishes **initializing the global data structures**, and **"creates" the other tasks** in the system. UserMain can carry on executing at a low scheduling frequency. It might be used later on to shut down the system in a safe way.

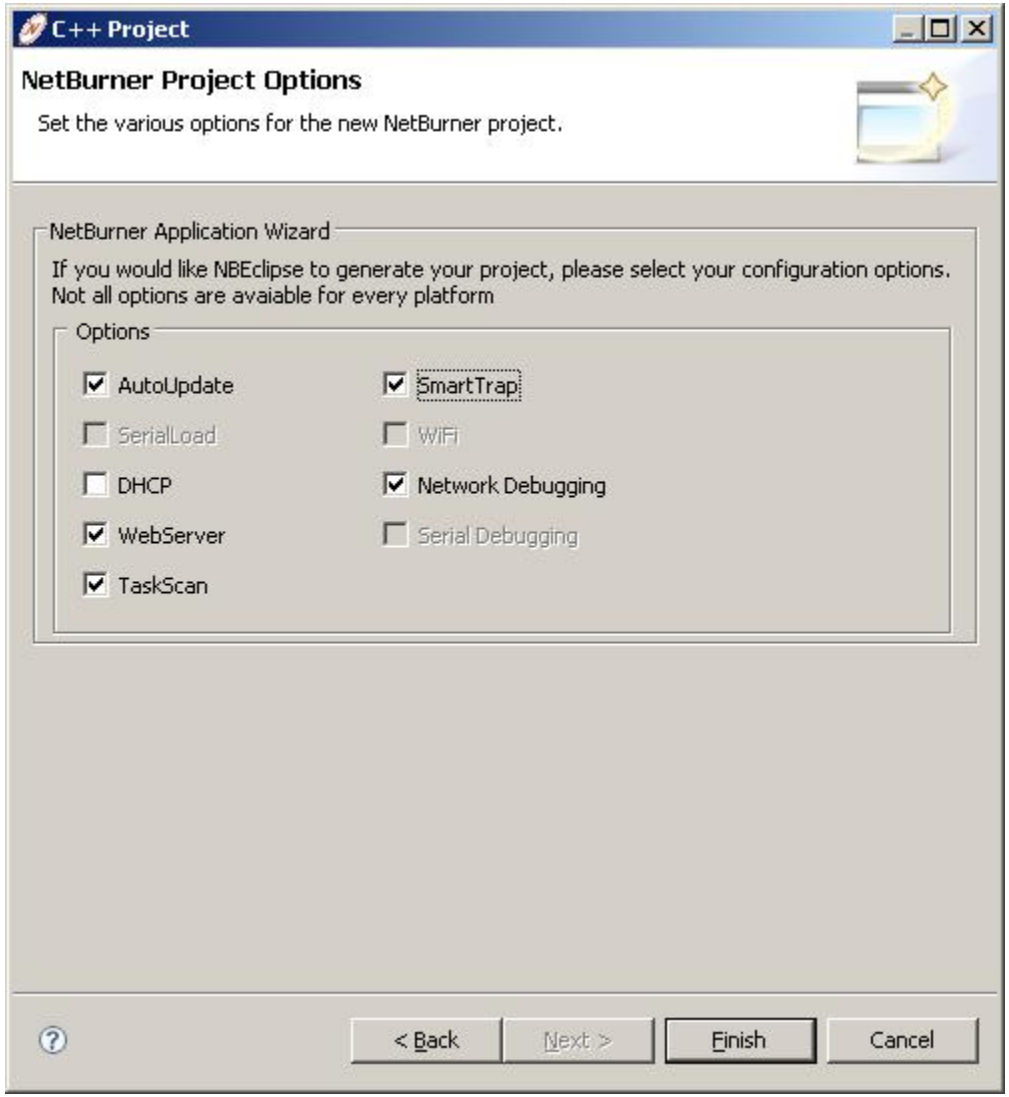
“Hello World!” Example, Part 1

```
#include "predef.h"          // constants used during debugging
#include <stdio.h>           // standard input/output functions
#include <ctype.h>           // useful type definitions
#include <startnet.h>        // TCP_IP and HTTP start-up functions
#include <autoupdate.h>      // for downloading system builds to flash
#include <dhcpclient.h>      // create DHCP client to get IP address
#include <smarttrap.h>       // to allow smart traps
#include <taskmon.h>         // to access task monitor features
#include <NetworkDebug.h>    // to use networked debugger

extern "C"   // required for plain C function calls from C++ code
{
    void UserMain(void *Pd);  // declare main task function prototype
}

                              // define application name for NBEclipse
const char *AppName="Hello_World_application";
```

“Hello World!” Example, Part 2

```
void UserMain(void *Pd) {           // No input argument here
    InitializeStack();              // init the TCP_IP stack
    if (EthernetIP == 0) GetDHCPAddress();
            // get IP address if not already allocated statically
    OSChangePrio(MAIN_PRIO);        // initialize task priority
    EnableAutoUpdate();             // enable auto update to flash
    StartHTTP();                    // start web server
    EnableTaskMonitor();            // enable task scan monitor

    #ifndef _DEBUG
    EnableSmartTraps();             // use smart traps by default, or
    #endif
    #ifdef _DEBUG
    InitializeNetworkGDB_and_Wait();  // use networked GDB debugger
    #endif

    while (1) {                     // loop forever
        iprintf("Hello World!\n"); // a version of printf
        OSTimeDly(TICKS_PER_SECOND * 1);  // wait 1s per iteration
    }
}
```

NBEclipse Application Wizard Settings



MicroC/OS System Files

`ucos.c`	Generic MicroC/OS function definitions
`ucosmain.c`	Start-up and debug functions, and the stack definition for the **`IdleTask()`**
`ucosmcfc.c`	ColdFire-specific MicroC/OS C functions, such as OSTaskCreate, Task Control Block init, and task stack checking code.
`ucosmcfa.s`	ColdFire-specific functions in assembly language for task switching and the timer.
`main.c`	Contains **`main()`** function that does system initialization and creates **`UserMain()`**.

Creating MicroC/OS Tasks (1)

- The NetBurner port creates the **`UserMain()`** task.
- Additional application tasks can be “created” within **`UserMain()`** using the **`OSTaskCreate`** function.
- The code for the application tasks must be present in the source file before **`UserMain()`** so that it is compiled and ready to execute.

```
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <ucos.h>
#include <dhcpclient.h>

extern "C"    // Allow old-style C function prototypes
{
  void MyTask( void *Pd );
  void UserMain( void *Pd );
}
```

Creating MicroC/OS Tasks (2)

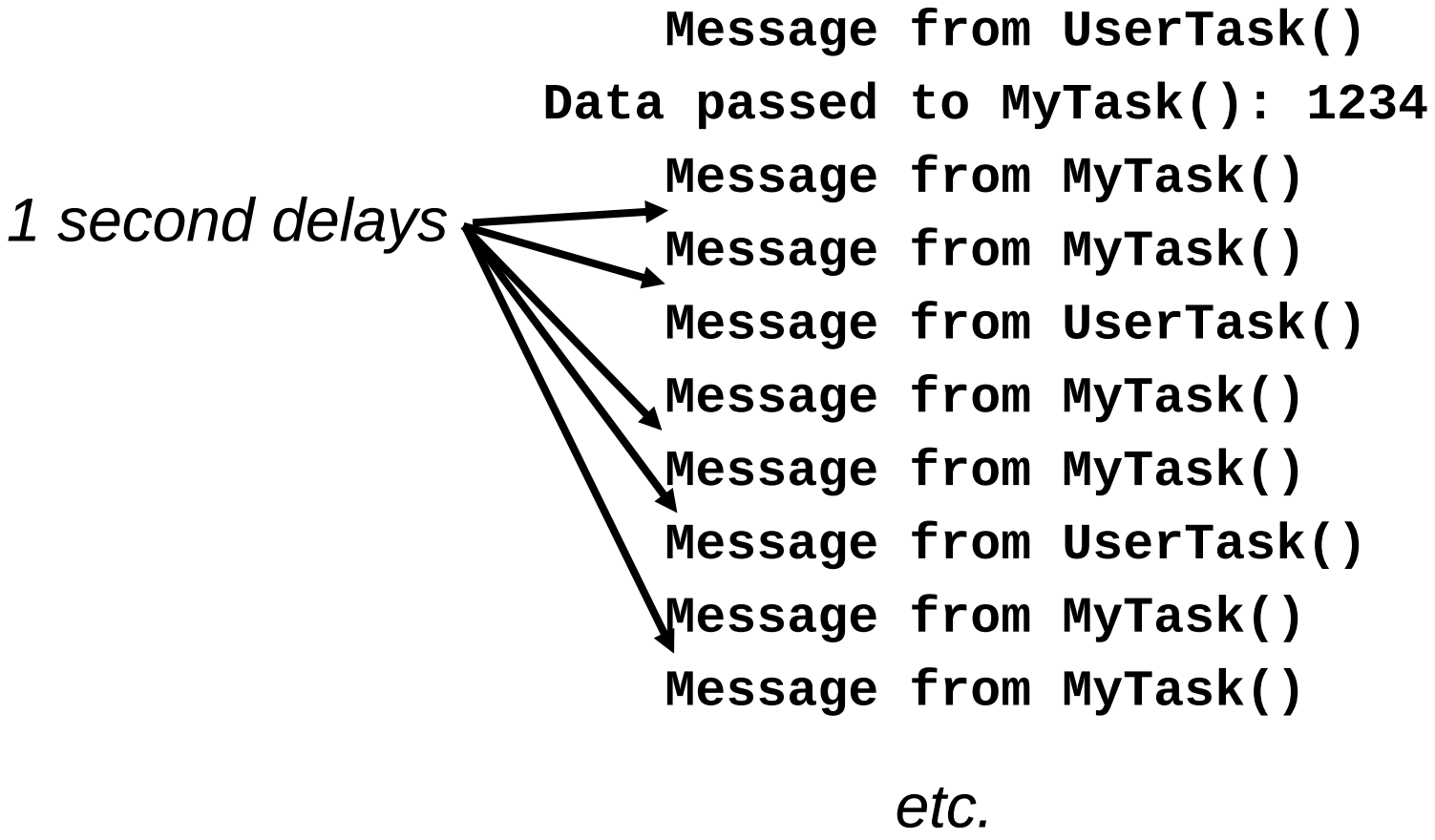
```
asm( " .align 4 " );     // Go to next 4-byte-aligned address to
                         // ensure that the stack is 4-byte-aligned

// Global data structure definition for the new task's stack
DWORD MyTaskStack[USER_TASK_STK_SIZE]
                    __attribute__( ( aligned( 4 ) ) );;

void MyTask( void *Pdata )
{
   WORD data = *(WORD *)Pdata; // cast the passed parameter

   iprintf( "Data passed to MyTask(): %d\r\n", data );

   while ( 1 )
   {
      iprintf( "    Message from MyTask()\r\n" );

      OSTimeDly( TICKS_PER_SECOND * 1 );  // one second
   }

}  // end of MyTask
```

Creating MicroC/OS Tasks (3)

```
void UserMain( void *Pd )
{
   InitializeStack();
   OSChangePrio( MAIN_PRIO );
   if ( EthernetIP == 0 ) GetDHCPAddress();
   EnableAutoUpdate();

   WORD MyTaskData = 1234;

   if ( OSTaskCreate ( MyTask, (void *) &MyTaskData,
            (void *) &MyTaskStack[USER_TASK_STK_SIZE],
            (void *) MyTaskStack, MAIN_PRIO+1 ) != OS_NO_ERR )
   {
      iprintf( "*** Error creating MyTask\r\n" );
   }

   while ( 1 )   // Could have !shutdown_system condition here
   {
      iprintf( "    Message from UserTask()\r\n" );
      OSTimeDly( TICKS_PER_SECOND * 2 );  // two seconds
   }

   // Could have system shut-down commands here
}  // end of UserMain
```

Creating MicroC/OS Tasks (4)

Resulting text output:



MicroC/OS functions that can cause task blocking

MicroC/OS calls:

OSTimeDly()

OSSemPend()

OSQPend()

OSMboxPend()

I/O System Calls:

`select()`
`read()`
`write()`
`gets()`
`getchar()`
`fgets()`

Network Calls:

`accept()`
`Also, creation of UDP packet`

Functions that can cause task unblocking

MicroC/OS calls:

OSTimeTick() – called by the timer ISR

OSSemPost() – called by the running task or an ISR

OSQPost() – called by the running task or an ISR

OSMboxPost() -- called by the running task or an ISR

Predefined Data Types in MicroC/OS

Data types available in generic MicroC/OS and MicroC/OS-II:

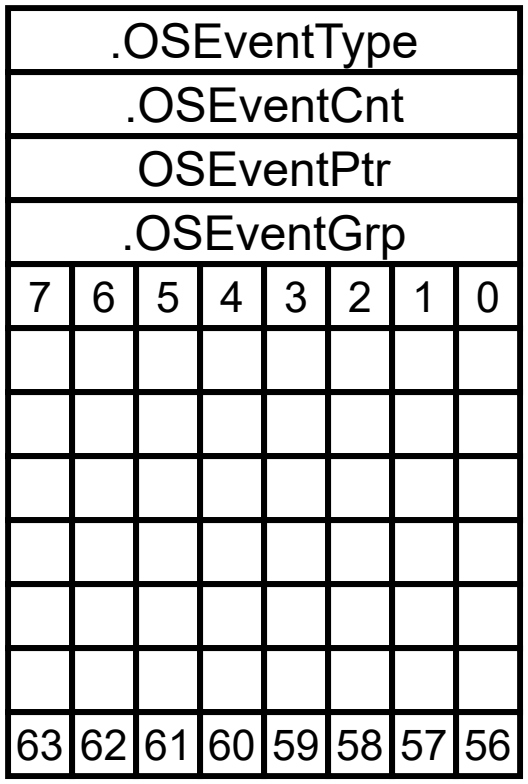
OS_TCB	Data type for Task Control Blocks
OS_EVENT	Data type for Event Control Blocks
OS_SEM_DATA	Data type used to return semaphore data

Other symbolic type definitions in the NetBurner ports:

BYTE	ColdFire 8-bit byte
WORD	ColdFire 16-bit word
DWORD	ColdFire 32-bit longword. Preferred storage type for fast access. Should be 4-byte aligned.
BOOL	Boolean
OS_SEM	Semaphore data structure
OS_FIFO	First-In First-Out queue data structure
OS_MBOX	Mailbox data structure
UINT8	Unsigned 8-bit integer
UINT16	Unsigned 16-bit integer
UINT32	Unsigned 32-bit integer

See C:\Nburn\includes\basictypes.h for all of the basic types.

Event Control Blocks (ECBs)



- MicroC/OS-II uses Event Control Blocks to record information that is associated with objects that possibly have blocked tasks.
  e.g. semaphores, mutual exclusion semaphores, mailboxes, queues
- ECBs are not used in NetBurner MicroC/OS
- User tasks do not normally look inside these ECBs, but tasks often need to handle pointers to ECBs when they call MicroC/OS-II functions.
- Field .OSEventTbl[] contains 64 bit flags that record which tasks are currently blocked while waiting on the associated object.
- Field .OSEventCnt gives the total number of tasks that are blocked waiting on the object.

Function Return Codes in Netburner MicroC/OS

All of these symbolically-defined codes are of type BYTE.

OS_NO_ERR	No error occurred.
OS_TIMEOUT	Timeout expired.
OS_MBOX_FULL	Mailbox is full.
OS_Q_FULL	Queue is full.
OS_PRIO_EXIST	Task ID already in use.
OS_SEM_ERR	Invalid priority supplied.
OS_SEM_OVR	Over max. semaphore count.
OS_CRIT_ERR	Critical OS error.
OS_NO_MORE_TCB	No more TCBs available.

Task Creation

BYTE **OSTaskCreate**(void (* Ptask) (void * Ptaskfunc), void * Pdata,
void * Pstacktop, void * Pstackbot, BYTE priority);

- Creates a new user task.
- "Ptask" points to the first instruction in the task.
  Note: This argument is just the name of the task function.
- "Pdata" points to an optional task-specific input argument.
- "Pstacktop" points to the top of the task's private stack.
  Note: The stack must be "4-byte-aligned" in memory.
- "Pstackbot" points to the bottom of the task's private stack.
- "priority" is a unique identifier from 0 to 63.
- Return code is OS_NO_ERR if task created successfully.
- Return code is OS_PRIO_EXIST is priority is already in use.

Example of Task Creation

```
asm( " .align 4 " );    // go to next 4-byte aligned address

// Reserve memory for the task MyTask's stack
DWORD Pstacktop[USER_TASK_STK_SIZE]
                          __attribute__( ( aligned( 4 ) ) );

void MyTask ( void * Pdata )
{
    // task code goes here
}

void UserMain ( void * Pd )
{
    // beginning part of task UserMain

    if (OSTaskCreate( MyTask, (void *) Pdata,
                (void *) &Pstacktop[USER_TASK_STK_SIZE],
                (void *) Pstacktop, MyTaskPriority )
                     != OS_NO_ERR) {
          // handle task creation error
    }
    // rest of UserMain
}
```

Task Deletion

void **OSTaskDelete**(void);

- Moves the calling task into the dormant state.
- A deleted task can be re-created by another task by calling OSTaskCreate.
- The code for the task, and usually also the task's stack area in memory, are left alone.

Changing a Task's Priority

BYTE **OSChangePrio**(BYTE newpriority);

- Used to change the priority of the calling task.
- Can't be called directly by other tasks.
- Priority "newpriority" must not already be in use.
- Return code is OS_NO_ERR if successful.
- Return code is OS_PRIO_EXIST if newpriority is already assigned to an existing task.

Task Delay Management Functions

void **OSTimeDly**(WORD ticks)

- Block the calling task by "ticks" timer increments.
- The highest available ready-to-run task starts executing.
- The tick unit is defined from TICKS_PER_SECOND.
- The ECE 315 lab system has 200 ticks per sec.
- The largest possible delay is 65,535 ticks.
- There is no return code.

void **OSChangeTaskDly**(WORD priority, WORD newticks)

- Cancel the time delay for another task, which has priority "priority" and that is currently waiting, and change it to the "newtick" delay.
- There is no return code.

Semaphore Management Functions

BYTE **OSSemInit**(OS_SEM * Psem, long count)

- Initializes counting semaphore pointed to by "Psem".
- The initial long count value is set to "count".
- Returns pointer to semaphore data structure.
- Return code is OS_NO_ERR if successful.
- Return code is OS_SEM_ERR if the requested count is less than zero.

```
OS_SEM MySemaphore;    // create semaphore area

OSSemInit( &MySemaphore, 0 );  // initialize sem
```

Semaphore Management Functions

BYTE **OSSemPend**(OS_SEM * Psem, WORD timeout);

- Block and wait on the semaphore pointed to by "Psem".
- If the semaphore count is greater than zero, then the count is decremented by one and the task proceeds.
- If the semaphore count is zero or less, then the count is decremented and the task is moved to a blocked queue associated with "Psem", and the highest ready-to-run task begins executing.
- "timeout" sets the maximum block time in ticks.
- if timeout == 0, then an infinite wait is assumed.
- Return code is OS_NO_ERR if successfully unblocked before the timeout expired; otherwise get OS_TIMEOUT.

```
OS_SEM MySemaphore;              // before all tasks
OSSemInit( &MySemaphore, 0);     // in UserMain task

OSSemPend( &MySemaphore, 100 );     // in another task
```

Semaphore Management Functions

BYTE **OSSemPost**(OS_SEM * Psem);

- Signal (increment) a semaphore pointed to by "Psem".
- The highest priority task waiting on semaphore "Psem" (if any) is moved to the ready-to-run queue.
- Return code is OS_NO_ERR if successful.
- Return code is OS_SEM_OVF if the value of the semaphore overflowed as a result of the post.

```
OS_SEM MySemaphore;                 // before all tasks

OSSemInit( &MySemaphore, 0);        // in UserMain task

OSSemPost( &MySemaphore );          // in another task
                                    // or in an ISR
```

Message Queue Management Functions

BYTE **OSQInit**(OS_Q * Pqueue, void **Pqstart, BYTE QSIZE);

- Initialize a circular queue of pointers to data elements.
- "Pqueue" points to an OS_Q object that keeps track of the queue of tasks that are ever blocked on the queue.
- "Pqstart" is a pointer that points to an array of pointers to (void *)'s. Each (void *) points to a data element, such as a character array.
- Must previously have allocated memory for the array of (void *)'s by declaring the array “void * Pqstart [QSIZE];”
- "QSIZE" is the maximum number of pointers in the queue.

```
OS_Q MyQueue;                          // before all tasks
void * MyQueueArray[ QSIZE ];

OSQInit( &MyQueue, MyQueueArray, QSIZE ); // UserMain
```

Message Queue Management Functions

void * **OSQPend**(OS_Q *Pqueue, WORD timeout, BYTE * Perr);

- Block and wait on the queue pointed to by "Pqueue".
- "timeout" gives a maximum wait time in ticks.
- A "timeout" of 0 means that an infinite wait is possible.
- Return code pointed to by "Perr" is either OS_NO_ERR or OS_TIMEOUT.
- The return value of the function is a pointer to the next data element at the head of the queue.
- If the queue was empty before the function call, then the return value is the predefined C constant NULL.

Message Queue Management Functions

BYTE **OSQPost**(OS_Q * Pqueue, void * Pmsg);

- Deposit the message pointer "Pmsg" into the message queue pointed to by "Pqueue".
- OS_NO_ERR is returned if the message got posted successfully to the queue.
- Otherwise, the return code has value OS_Q_FULL if the message queue is full and it has no room to hold the new message pointer.

```
OS_Q MyQueue;              // in UserMain task
void *MyQueueArray[ QSIZE ];
OSQInit( &MyQueue, MyQueueArray, QSIZE );

char *Pmsg;                // in another task
Pmsg = "Hello World!";
OSQPost( &MyQueue, (void *) Pmsg );
```

Temporarily Disabling Hardware Interrupts

void **USER_ENTER_CRITICAL**( );

- Sets the interrupt mask bits in the CPU status register to 111 to block all maskable interrupts. The previous mask bit pattern is saved on the supervisor stack.
- All MicroC/OS functionality is disabled.
- Each call to USER_ENTER_CRITICAL must be matched with exactly one subsequent call to USER_EXIT_CRITICAL
- This macro is used at the start of critical sections within many other MicroC/OS functions (e.g., OSSemInit, OSSemPend, OSSemPend, the message queue functions, etc.).

void **USER_EXIT_CRITICAL**();

- Restores the interrupt mask bits to what they were immediately prior to the previous call to USER_ENTER_CRITICAL.
- This macro is used within many MicroC/OS functions to end critical sections that were begun with USER_ENTER_CRITICAL.

Temporarily Disabling Multitasking

void **OSLock**(void);

- Prevents task context switches and pre-emption according to task priority, but does not otherwise affect interrupt handling or access to MicroC/OS functions.
- Each call to OSLock must be matched with exactly one subsequent call to OSUnlock.

void **OSUnlock**(void);

- Restores normal multitasking.
- The call to OSUnlock will immediately trigger a context switch if a ready-to-run task other than the calling task now has the highest priority in the multitasking environment.

MicroC/OS Configuration

- MicroC/OS-II can be configured readily at compile time using "switches" in files “OS_CFG.h” and “includes.h”.
- Unnecessary object code for unused functions can be omitted from the system to minimize memory space.
- The NetBurner version of MicroC/OS has only three switches in "nburn/includes/predef.h"

```
#define UCOS_STACKCHECK 1     /* provide OSDumpTCBTasks() */
                              /*      and OSDumpTasks()   */
#define UCOS_TASKLIST   1     /* provide ShowTaskList()   */
#define BUFFER_DIAG     1     /* provide ShowBuffers(),   */
                                /* GetBufferX and FreeBufferX  */
                             /* see C:\Nburn\includes\buffers.h */
```

Interrupt Handling in MicroC/OS

Interrupt Service Routines (ISRs) can be coded in assembly language using the following pattern:

1) Save all CPU registers on the interrupted task's stack
2) Increment the system variable OSIntNesting
3) If OSIntNesting == 1 then OSTCBCur->OSTCBStkPtr = SP
4) Clear the interrupting device
5) (optional) Re-enable interrupts to allow IRQ nesting
6) Execute the code that services the interrupt request. This code should execute quickly, and must not be blocked.
7) Call OSIntExit() to check for possible task switch after RTE
8) Restore all CPU registers
9) Execute an RTE instruction

Interrupt Handling using the INTERRUPT Macro

INTERRUPT(ISR_name, NewSRValue)

- INTERRUPT wraps an Interrupt Service Routine (ISR) with the necessary code for saving the CPU registers at the start of the ISR, and then restoring the CPU registers at the end of the ISR.
- At the end of the ISR, the possibility of a context switch is also checked by MicroC/OS.
- The UserMain task must correctly load the corresponding exception vector entry with the address of the first instruction of the ISR.

Example:

```
// The vector value &eTPU_ISR must be written during initialization
// into the eTPU vector location in the Exception Vector Table.
INTERRUPT( eTPU_ISR, 0x0500 )
{
    // ISR code for the eTPU goes here
    // All interrupts at level 5 and below are masked out in this ISR
}
```

ISR Execution Can Cause a Task Switch

- When an ISR executes, it can unblock tasks that were waiting on flags, mailboxes, queues, semaphores, etc.
- The originally interrupted task may now have a lower priority than one of the unblocked ready-to-run tasks.
- The function OSIntExit, which is called by code inserted by the INTERRUPT macro, checks for this possibility.
- OSIntExit either reloads and schedules the interrupted task (if it is still the highest priority task that can run), or it causes a context switch to the new highest priority ready-to-run task.