

OPEX ... OPERATIONS EXECUTIVE

**A Powerful And Easy To Use
Control Program
For Microprocessors**

**Version 1.0
(c) 2004 Steve Childress
steveh@san.rr.com**

What Is OPEX? A document roadmap in [this table](#).

OPEX is a simple to use software **Operation Executive** for microprocessor based control software which is time and event based. OPEX also includes commonly needed utility functions such as a clock/calendar, a task scheduler, flag/event wait/wakeup, buffered serial I/O, and task monitoring/debugging aids. A demonstration program shows these, plus Dallas 1-Wire I/O and push-button debouncing for event triggered actions. A document roadmap in [this table](#).

It's an RTOS?

Not in the preemptive multitasking sense. OPEX is an for applications where there are multiple software tasks to be run at fixed or variable time intervals, at a given date and time, or upon a particular interrupt or serial I/O event. When run, each task chooses when it runs next, or it quits. A task may suspend itself until a certain inter-task signal/event occurs, such as the arrival of serial port data or a flag is set/cleared by a user-written or OPEX interrupt service routine (ISR).

Is it memory-efficient?

Many or most task oriented programs can be arranged to not need preemption. OPEX uses a single stack for all tasks – a major savings in time and RAM. Minus the OPEX demo program, the debugging aids and the C library, OPEX itself is a relatively small amount of code. A 16KB FLASH ROM is probably the minimum. Each task needs only about 22 bytes of RAM for status and state, and this is user-extensible for more state information.

Is assembly language guru-ism needed?

No, OPEX is 100% C code, with the microprocessor-specific code in a user-alterable file.

What's the multitasking strategy?

Most often, when there are multiple tasks an application, these can be either (1) written as each as a state machine, suspending itself for an event or a time delay between states thus avoiding looping, or (2) written as an event driven task, suspending itself until an I/O event or an inter-task signal occurs. Done properly, these techniques avoid the need for a full preemptive multitasking RTOS which is costly in RAM usage (a stack per task) and context switching times. OPEX is a structured form of cooperative multitasking, but with an emphasis on running well behaved tasks at the planned time or after an expected interrupt or an inter-task signal.

What's the task scheduling time granularity?

It's whatever clock interrupt rate you choose. A typical number would be $1/128^{\text{th}}$ or $1/8^{\text{th}}$ of a second. So a task may scheduled to run, say, every $1/128^{\text{th}}$ of a second, every half-second, every minute, etc. Or at a given date and time. Or upon an I/O or interrupt event, or an inter-task signal.

How are tasks created and terminated?

At run-time, any task may create another instance of itself or another task. Any task may terminate (kill) any other task whose name or address is known, including tasks waiting on an event. OPEX has an internal memory (RAM) allocator/deallocator – derived from malloc. To avoid memory “leaks”, OPEX automatically frees allocated memory if a task quits without doing so or is terminated by another task.

There's a clock and calendar?

Yes, there's a software clock/calendar with leap year, daylight saving correction logic.

What about certain code which is more time-sensitive?

OPEX calls a low-overhead user-supplied “idle” function whenever there are no tasks ready to run. If tasks don't hog the processor, the idle function can run 100,000 or more times per second. Or, an interrupt routine can call OPEX to wake up a suspended task and run it as soon as the interrupted task suspends itself.

For more specifics and examples, read on ...

CONTENTS (*click on hyperlinks below*)

OPEX - OVERVIEW	5
To “RTOS”, or Not.....	6
OPEX SERVICES OVERVIEW.....	7
EXAMPLES	8
INTRODUCTORY EXAMPLE PROGRAM.....	9
OPEX Scheduler Status Display Example	10
Task Scheduling.....	13
Task Techniques: State Machines	15
OPEX HARDWARE ENVIRONMENT.....	16
OPEX FLASH MEMORY SIZE	17
OPEX RAM SIZE	17
OPEX EEPROM SIZE.....	18
OPEX Function Programming Reference	19
INITIALIZATION	19
TASKS.....	20
DATE / TIME	23
SERIAL OUTPUT	25
SERIAL INPUT	27
MONITORING AND DEBUGGING	28
OPEX Data Structures.....	30
OPEX DATE_TIME STRUCTURE.....	30
OPEX TASK CONTROL BLOCK (OPEX_TCB) STRUCTURE	30
OPEX QUEUES	30

Document Roadmap, Topic Hyperlinks (click on these)

[OPEX OVERVIEW](#)

[To RTOS or Not](#)

[OPEX Services Overview](#)

[PROGRAM EXAMPLE #1](#)

[HARDWARE ENVIRONMENT AND MEMORY](#)

[OPEX FUNCTION OVERVIEW](#)

[SCHEDULER DISPLAY EXAMPLE](#)

[OPEX FUNCTIONS](#)

[Initialization](#)

[Tasks](#)

[Date/Time](#)

[Serial Output](#)

[Serial Input](#)

[Monitoring and Debugging](#)

[OPEX Task Control Block Structure](#)

[OPEX Queues](#)

[OPEX Date/time structure](#)

OPEX - OVERVIEW

(see also [“Hardware Environment”](#), [“To RTOS or Not”](#), and [OPEX Services Overview](#), below).

OPEX Is ...

C software	For an embedded system (microprocessor) programmer who is using the C language.
A control program	To oversee multiple “thread-like” processes. Each process or “task” uses only ~22 bytes of overhead
A Scheduler	<p>To cause processes to run when desired, such as:</p> <ul style="list-style-type: none"> • At specific dates and/or times, optionally recurring • At time intervals, either fixed or variable, optionally recurring • When the serial port receives a byte or an end-of-line character • When inter-process signals are received
An Easy Way To Implement Multiple State Machines	<p>Using the Scheduler’s functions which include, among others</p> <ul style="list-style-type: none"> • Task Create and Quit • Task time delay or wait for specific date/time • Task wait-for-flag, set-flag (semaphore)
A clock and calendar	With time zone, daylight savings and leap year, maintained using a microprocessor timer such as the 32768KHz asynchronous oscillator of the Atmel megas.
Serial port support	Functions for interrupt driven input/output, optionally non-blocking, for binary or string (ASCII 8 bit) data
Monitoring and Debugging Aids	Functions to format and display a list of all processes, the detailed state and status of each, plus RAM used and unused, and maximum observed stack depth
Automatic Memory Allocation Housekeeping	A task may use the OPEX forms of malloc() to obtain temporary use of a block of RAM. When the task quits, OPEX automatically frees this memory if the task did not do so. A modified version of the standard malloc() is included in OPEX.
Complete with Examples	<p>Dallas 1-wire reader for DS1820 temperature chips</p> <p>I/O bit sensing and switch debouncing</p> <p>Serial I/O for host commands and set clock/calendar</p>

To “RTOS”, or Not

A Real Time Operating System (RTOS) arguably is an overkill for many microprocessor applications. Preemptive multi-tasking, a hallmark of RTOSes, is a convenient abstraction for the programmer, but it comes with a high cost in RAM usage: one stack per task, and each stack must be large enough for the worst case nested function calls and interrupts. Task switching is also burdensome due to changing stacks and CPU conditions per task.

OPEX, using one stack, works well if there are several independent processing tasks to do, some recurring, some rarely done. Of course, OPEX allows that classic small amount of highly time critical code to be an interrupt service routine, while the rest of the application tasks run at specific dates and times, or on a recurring basis, on receipt of an inter-task signal, or on arrival of serial port data.

Tasks in OPEX are created and destroyed at run-time. This is unlike some RTOSes which have a fixed, compile-time arrangement of tasks.

The OPEX scheduler clock tick time increment can be changed easily. The default choice is $1/128^{\text{th}}$ of a second. With a 6MHz or faster CPU, this is quite a few CPU cycles per task invocation. A rate of $1/8^{\text{th}}$ second may fit some applications better. And again, in many cases, no harm is done if a task occasionally uses a bit more than one time slice.

If the application tasks can each do their work, or a portion of the work in one scheduler clock tick time, then all tasks will be run at precise times. Most tasks can be coded to avoid long loops, or to relinquish the CPU to other tasks for a bit of time. One common method is to code a task as a state machine. When re-run, the task's variables and state is preserved by OPEX. If a task occasionally runs a bit longer than a clock tick, then the next-waiting task might be delayed.

An “idle” task in OPEX can be established and changed at run-time. This is not a task per se. It is a C function which is called repeatedly, and should return quickly, until a task per se is ready to run. On a 6MHz AVR, a simple idle task is called about 100,000 times per second. Thus, it can monitor I/O activities which don't interrupt.

OPEX then is a framework for cooperative multi-tasking – a leaner way to do things, which also permits that small percentage of time critical code to run at the interrupt level.

The examples, below, show the power of the date/time/event scheduling, where a task may arrange itself to run every x days/hours/minutes/seconds/ticks, or at a specific date/time, or upon an inter-task signal or receipt of serial data.

OPEX SERVICES OVERVIEW

Tasks:	<i>Create a task, obtaining its status block address. Reschedule self to run at a time and date or run after a time delay Task A may reschedule when Task B will run next Task A may wait on s signal from Task B or C or D, ... Task A may “quit” or may “kill” task B (kill task) Get a task’s status block address given the task’s name Use an “idle” task to run when no other task is ready to run.</i>
Events	<i>Wait for any of n events which are each bits in a user defined byte Set or clear an event bit or bits in a given byte. This will cause all tasks waiting on that event bit to change to be rescheduled. An event may be created by a task or by an interrupt service routine.</i>
Date/Time	<i>Maintain a clock/calendar with daylight savings and leap year. Get and set the date and time and time zone (GMT offset) When the date/time is changed, all tasks in the schedQ are immediately run to permit themselves to react an reschedule. Compare two date/time data items to determine <, =, > Calculate day of week for a date Format a date/time to a string</i>
Serial I/O	<i>OPEX provides functions to send and receive text and binary data. A task may wait for the arrival of a serial port character or an ASCII CR Transmission uses buffered and interrupt driven output. To avoid blocking if the output buffer is full, a task may retrieve the buffer count and elect to reschedule itself to run later, thereby not holding up other tasks. Reception uses interrupt driven buffering. OPEX has functions to wait-for-any-byte received, wait-for-CR, get a byte with a timeout, and get a string ending with a CR. A “peek” function also allows a task to determine if data is available.</i>
Monitoring	<i>Display (to serial port) a list of all task status (see below) Optionally, each task may elect to display its status as it runs.</i>
Examples	<i>A set of code provides samples of most OPEX services. An interactive, serial port based command/response processor with Parser for setting the date/time from the serial port Examples of push-button debouncing A Dallas 1-wire DS1820 temperature sensor example</i>

EXAMPLES

The examples below are introductory.

Additionally, in the distribution files there is a complex example program showing multiple tasks, including push-button I/O, Dallas 1-Wire net I/O (DS1820 temperature sensors), and LED manipulation.

INTRODUCTORY EXAMPLE PROGRAM

This small program has one task which prints a message once a second using the [OPEX Serial I/O Services](#). The main() in this program initializes things, creates one task, and invokes the scheduler which runs forever or until all tasks quit. This source code is compiled with the file AVRdependent.c.

The program's output on the serial port is:

```
Starting Scheduler
20:55:16.06 20:55:16.00 Wed 24 Mar 04 TASK0 @0x2A6 state=0x0 info=0x0
TASK0: pinfo = 0000
20:55:17.06 20:55:17.00 Wed 24 Mar 04 TASK0 @0x2A6 state=0x0 info=0x1
TASK0: pinfo = 0001
20:55:18.06 20:55:18.00 Wed 24 Mar 04 TASK0 @0x2A6 state=0x0 info=0x2
TASK0: pinfo = 0002
```

```
////////////////////////////////////
// OPEXlib_Demo0.c - A very simple demo
#include "OPEX.h"
void serial_init(void); // these are in AVRdependent.c
void io_init(void);
void timer_init(void);
char StringBuf1[128]; // global scratch buffer for string work

// THE ONE SCHEDULED TASK's FUNCTION //////////////////////////////////
// say hello, then reschedule myself to run again later
void task0(OPEX_TCB *me)
{
    OPEX_task_report(StringBuf1, me); // monitor/debug report
    sprintf_P(StringBuf1, PSTR("%s: pinfo = %04d"), me->name, (int)me->pinfo++);
    OPEX_putline(StringBuf1);
    OPEX_sched_resched(me, 0, 0, 0, 1, 0); // run me again later
}
/// MAIN ///
int main(void)
{
    io_init(); // see AVR_dependent for these...
    timer_init();
    serial_init();
    OPEX_com_init(); // init OPEX serial I/O
    sei();

    OPEX_init_dateTime(); // set OPEX date and time defaults
    OPEX_set_monitoring(1); // enable debug monitoring

    for ( ;; ) {
        OPEX_putline_P(PSTR("Starting Scheduler") ); // print from flash memory
        OPEX_sched_new(&task0, "TASK0"); // launch task, assume success doing so
        OPEX_sched_start(); // returns if all tasks quit
    }
}
```

OPEX Scheduler Status Display Example

A quick way to understand what OPEX can do is to read the OPEX scheduler's status display. The sample, below, was produced by the [OPEX Monitoring and Debugging Services](#). The column "Ref." reference letters A, B, ... refer to the explanations below the figure. This display is from a Mega32 chip running a test program with 13 dissimilar tasks. See also [Task Scheduling Basics](#).

Ref.

```

A      unused: 418 stack depth max: 154, SP lowest: 0x7C5 malloc next: 0x623
B      free list: 22@0x593, 70@0x5DB, Total:92, num_serial_blocks: 2070
C      22:54:20.50 Sat 13 Mar 04 DST=0 GMT offset std: -8 GMT offset_now: -8
        schedQ:
D      1: 22:54:20.43 Sat 13 Mar 04 <noName> @0x51D state=0x0 info=0x0
        2: 22:54:20.50 Sat 13 Mar 04 TASK8 @0x565 state=0x0 info=0x2
        3: 22:54:20.50 Sat 13 Mar 04 TASK9 @0x57D state=0x2 info=0x0
        4: 22:54:20.50 Sat 13 Mar 04 TASK6 @0x535 state=0x0 info=0x0
        5: 22:54:21.00 Sat 13 Mar 04 TASK1 @0x4BD state=0x0 info=0x0
        6: 22:54:21.50 Sat 13 Mar 04 TASK2 @0x4D5 state=0x0 info=0x0
        7: 22:54:25.00 Sat 13 Mar 04 TASK3-0 @0x5AD state=0x0 info=0x0
        8: 22:54:30.06 Sat 13 Mar 04 TASK0 @0x4A5 state=0x0 info=0x0
        9: 22:54:30.12 Sat 13 Mar 04 Task5x @0x5C5 state=0x0 info=0x2
        10: 22:54:30.56 Sat 13 Mar 04 TASK4 @0x505 state=0x0 info=0x4F1
        11: 02:00:00.00 Sun 04 Apr 04 DST @0x475 state=0x0 info=0x0
        flagwaitQ:
E      1: 22:54:20.37 Sat 13 Mar 04 host_cmd1 @0x48D state=0x0 info=0x0
        2: 23:31:46.00 Fri 12 Mar 04 TASK7 @0x54D state=0x0 info=0x0

```

Ref. Explanation

A unused: 418 stack depth max: 154, SP lowest: 0x7C5 malloc next: 0x623

unused RAM – the unallocated “heap” and unused stack space in RAM.

stack depth max. is sampled by the clock interrupt and saved.

SP lowest is the RAM address associated with *stack depth max*.

B free list: 22@0x593, 70@0x5DB, Total:92, num_serial_blocks: 2070

OPEX v1.0

free list is a list of blocks of previously allocated RAM which are now available for reuse.

The *free list* here shows two blocks, one of 22 and one of 70 bytes. A newly created task's status will use RAM from this list

Num serial blocks is a counter of the number of times the serial port output buffer was full when a task called to transmit more data. This is for debugging purposes. Blocking causes the task to "block" or loop and wait until space is available. This may cause somewhat tardy scheduling of other tasks. A task may avoid blocking by retrieving OPEX's unused serial output buffer count before transmitting, and time-delaying its transmission if need be.

C 22:54:20.50 Sat 13 Mar 04 DST=0 GMT offset std: -8 GMT offset_now: -8

Time and date (local time) of the status report.

Times are displayed as HH:MM:SS.nn, where nn is a fraction of a second, and the fraction relates to the microprocessor's timer interrupt interval.

Also shown are the time zone (GMT offset) as configured using the OPEX set time function. The DST indicator depicts if daylight savings is in effect. Daylight savings on/off is calculated by OPEX for North America and Europe.

D 1: 22:54:20.43 Sat 13 Mar 04 <noName> @0x51D state=0x0 info=0x0

schedQ a list of eleven tasks (processes) queued to run later.

Each task will run at the time and date shown for each, or somewhat later if a preceding task was tardy in completing its processing. The use of state machine techniques in OPEX allows tasks to avoid looping which would delay other tasks.

Each task optionally has a "name" which is given when the task is created. In this sample, most, but not all, tasks are named "TASKn". A task may change its name anytime, as is the case with TASK3-0 in the display. Note the task named "DST" is the task to change to or from daylight savings time when needed. A task's name is <noname> if a name string pointer is NULL (saves a bit of RAM).

To the right of the task's name is the RAM address of its task status data block.

The *state* field is a key variable used by tasks which are written as state machines, where *state* is used to select which code to run each time the task's date and time to run is met. This is the essence of how to avoid CPU-hogging while not requiring the overhead and high RAM usage of a preemptive RTOS.

OPEX v1.0

The *pinfo* field is another task variable. It may be used as anything, including a pointer to a larger block of RAM with application-specific task status data or buffers. In the sample, TASK4 has such. With this mechanism, there may be multiple instances of the same code, as different OPEX tasks.

E 1: 22:54:20.37 Sat 13 Mar 04 host_cmd1 @0x48D state=0x0 info=0x0

flagwaitQ is a list of two tasks queued and waiting for event(s).

Events are caused by (a) tasks or (b) an interrupt service routines (ISR). A task in the *flagwaitQ* is suspended, so obviously it cannot originate an event.

A task may wait on any of several events to occur. An event occurs when a task or ISR sets or clears a specific bit or bits in a particular “flag” byte. The particular byte is merely a RAM address (pointer) which the two tasks agree upon, by convention.

The OPEX serial port ISR has a particular byte with events for “a byte was received” and “an ASCII CR was received”. Using this, an OPEX task may queue and wait for input processing.

In this sample, “host_cmd” is a task waiting on serial data to arrive, and “TASK7” is waiting on an event triggered by another task every few seconds.

The time and date shown in the listing depict when the task was run, since it is waiting on an event, not a date/time.

Task Scheduling

New Tasks. When a task is initially created using `OPEX_sched_new()`, it is scheduled to run immediately after the current task completes (or, at initialization, after `OPEX_sched_start()` is called). Multiple instances of the same task code may exist, each with independent status and state.

When a newly create task runs, it should do whatever initialization is needed, then call either:

`OPEX_sched_resched()` to run again after a time delay, or

`OPEX_sched_schedAT()` to run again at a specific date and time, or

`OPEX_sched_on_flag()` to run when some other task or ISR accesses flag bit(s)

`OPEX_sched_quit()` if task instances is to cease to exist

Each time a task runs, it should do whatever processing is appropriate, make whatever OPEX service calls are needed, then make one of the above four OPEX calls. The task must then exit the task's C function promptly. If a task exits its function *without* calling one of the above, it will run again *after* all other tasks due at the same time.

As explained in the REFERENCE section, some OPEX calls do not return to the calling task, but rather, pass control to the scheduler.

Task Control Block (TCB). Each task has a small TCB in RAM. This is described at [OPEX Task Control Block Structure](#) .

Task Function Calling. Each time OPEX activates a task, execution begins at the **top** of the C function whose address is in the TCB. A task may alter this function pointer, or leave it unchanged, to change what function will run next.

Note: The TCB function pointer must point to a function with the standard task function prototype: `void foo(OPEX_TCB *)`.

If a task's TCB function pointer references function A, and A calls B, then B calls `OPEX_sched_resched` (or equivalent), then control goes to the scheduler. Note that B does not return to A. The next time the task is run, A will run again, given the function pointer in the TCB was not altered by A or B.

Task Termination. If a task calls

`OPEX_sched_quit()`

the task is abandoned.

A task may terminate any other task by passing that task's TCB address to

`OPEX_sched_kill()`

Task Techniques: State Machines

A state machine task, using a state vector stored in the TCB (see [OPEX TCB structure](#)):

```
void mytask(OPEX_TCB *me)
{
    switch (me->state) {
        case 0:
            // do whatever to initialize
            me->state = 1;
            break;
        case 1:
            // do whatever in state 1
            if (something)
                me->state = 2;
            break;
        case 2:
            // do whatever
            if (something_else)
                me->state = 1;
    }
    OPEX_sched_resched(me, 0, 0, 0, 0, 10); // run again in 10 clock ticks
}
```

Or by changing which function is to run when the scheduler reactivates the task:

```
void mytask1(OPEX_TCB *me)
{
    if (me->state == 0) {
        // initialize
        ++me->state;
    }
    if (something)
        me->funcp = &mytask1a; // new state, new function for this task
    OPEX_sched_resched(me, 0, 0, 0, 1, 0); // run again in 1 second
}

void mytask1a(OPEX_TCB *me)
{
    if (something_else_again) // if false, run same func again later
        me->funcp = &mytaskb;
    OPEX_sched_resched(me, 0, 0, 1, 0, 0); // run again in 1 minute
}

void mytask1b(OPEX_TCB *me)
{
    if (meaning_of_life != understood)
        me->funcp = &mytask1;
    else
        me->funcp = &mytask1a;
    // run in a half-second
    OPEX_sched_resched(me, 0, 0, 0, 0, TICKSPERSECOND/2);
}
```

OPEX HARDWARE ENVIRONMENT

The usual hardware on which OPEX can run is a microprocessor with 16K bytes or more of FLASH memory and 2KB or more of RAM, such as the mega32. The OPEX library can be compiled for the AVR chips which have less than 8KB of flash, however, these chips typically have only 1K of RAM which is marginal in having room for static variables and serial I/O buffers, but not impossible.

The AVR chip-dependent I/O is all in a single C file. The OPEX library typically does not need to be recompiled – only the chip-dependent code is changed for the chip to be used. The source code for this is easily changed and linked with the OPEX library and with the user's application program. The functions in the chip-dependent code include which serial port OPEX will use, the hooks for the interrupt handlers for the timer and USART/UART, and I/O initialization.

The serial I/O and clock interrupt entry points are in the chip-dependent code where OPEX library functions are called to service the interrupt. OPEX library routines also call functions in the chip-dependent C file to perform low level serial I/O. That is, the library calls code in AVRdependent.c which is compiled with the user's application.

The OPEX debugging and monitoring routines normally send output to a serial port. The chip dependent C file has the functions to hook and redirect this as desired, e.g., to an LCD.

OPEX FLASH MEMORY SIZE

The table, below, shows representative code sizes, as of this writing, for the modules within a small demonstration program. These sizes are with the compiler in the “default” optimization mode v.s. the “minimize code size” mode.

Code Size (bytes)	OPEX Module	Remarks
1522	OPEX.o	Scheduler and list management
1124	OPEXdateTime.o	Maintain date/time. Format date time for printing
336	OPEXdaylightSaving.o	Maintain/adjust daylight saving
910	OPEXserial.o	Buffered interrupt driven serial I/O with non-blocking
280	OPEXtimer.o	Maintains date/time and scheduler ticks
1030	OPEXUtils.o	Debugging and monitoring aids
5302	<i>SUBTOTAL</i>	
USER APPLICATION		
1220	AVRdependent	Includes optional code, e.g., save date/time to EEPROM
150	OPEXlib_Demo0	
STDLIB		
x	vprintf, etc.	
8860	<i>TOTAL, per the linker</i>	6668 with compiler in minimal code size optimization

OPEX RAM SIZE

Stack: The AVR chip’s stack is common to OPEX itself and to all tasks. Thus, the necessary stack size is no more so than for an ordinary program, i.e., large enough to accommodate nested function calls and interrupts in the application program.

Per instance of a task: Approx. 22 bytes plus optional per-task user data

OPEX static variables: Approx. 40

OPEX serial I/O buffers: Usually, the output buffer is set to 256 bytes to minimize blocking and the input buffer is 128 bytes.

OPEX EEPROM SIZE

None. The AVRdependent.c module has optionally-used routines to save and restore the date/time using EEPROM.

OPEX Function Programming Reference

INITIALIZATION

INITIALIZATION SERVICES	
Timekeeping initialization	<p>void OPEX_init_dateTime(void);</p> <p>The clock calendar is setup. Some applications may choose to retrieve a date/time from EEPROM or a hardware calendar chip.</p> <p>See "AVRdependent.c". In that file, other hardware devices are initialized such as the timer used by OPEX for scheduling and time keeping.</p>
Serial I/O Buffer Initialization	<p>void OPEX_com_init(void)</p> <p>Initializes the ring buffers within OPEX's buffered serial I/O. It also calls a serial port initialization routine within AVRdependent.c.</p>
Idle function (a function not coded to be an OPEX task)	<p>void OPEX_sched_setIdleFunc(void(*funcp))</p> <p>Establishes or redefines a C function to run when there are no OPEX tasks ready to run. This task should return promptly to keep latency low.</p> <p>The pointer argument is the idle function to use from this point forward. A NULL pointer may be passed to disable use of the Idle Task.</p> <p>This function is called after all tasks due to be run in the current time tick have run and returned, and until the next clock tick occurs. The rate at which the idle function is called implies the CPU utilization.</p> <p>The user function might include code to put the microprocessor to sleep until the next clock interrupt. Also, this function might provide watchdog timer support if needed.</p> <p>Note: The idle function itself must not call any OPEX functions. It should run and quickly exit in less than one clock tick.</p>
Scheduler start	<p>void OPEX_sched_start(void)</p> <p>Note: The scheduler is a function which will return when there are no tasks, i.e., none have been previously created or all have quit.</p>

TASKS

TASK SERVICES	
Idle Task	See Idle Function, above.
New task (create)	<p>OPEX_TCB * OPEX_sched_new(void (funcp)(OPEX_TCB *), char *)</p> <p><i>Purpose:</i> Create a new task.</p> <p><i>Returns</i> a pointer to the new task's TCB or NULL if the task could not be created (out of RAM). The return value need not be retained. The new task's TCB <i>state</i> and <i>pinfo</i> variables are initialized to zero, enabling the task to detect that it is a new instance. The new task is not run until the function that called OPEX_sched_new() returns.</p> <p><i>First argument:</i> a pointer to the C function which will run when the task first runs. This first-run function has this prototype: foo(OPEX_TCB *). The next function to run may be changed by the task as desired, as long as every function used has the defined prototype. Changing the "next function to run" is done by the task, altering a field in the TCB. A state machine can be implemented this way.</p> <p><i>Second argument:</i> a NULL or a pointer to a RAM-resident string which is the name of the task. The second argument may be quoted string (constant). See OPEX_sched_get_named().</p> <p>Note: For convenience, when the task's function is invoked by the scheduler, the task's TCB is passed as the argument to the function.</p>
Quit task	<p>void OPEX_sched_quit(void)</p> <p>The current task is abandoned. The TCB storage is freed.</p> <p>This function does not return; a jump to the scheduler is done.</p>
Kill task	<p>void OPEX_sched_kill(OPEX_TCB *)</p> <p>The task whose TCB address is passed as the argument is abandoned. The task may be on the schedQ (waiting for a date/time) or the flagwaitQ (waiting for an event).</p> <p>If a task passes its own TCB, the action is the same as for OPEX_sched_quit()</p>
Delay task	<p>void OPEX_sched_resched(OPEX_TCB *, days, hours, minutes, seconds, ticks)</p> <p><i>Purpose:</i> Reschedule a task to run after a time delay.</p> <p>If rescheduling itself, the calling task's function should promptly return();</p> <p><i>First argument:</i> a pointer to the TCB to be rescheduled which is usually the current task's TCB.</p> <p><i>Second to nth arguments:</i> the desired delay, declared as BYTE.</p> <p>Note: Minimal validity checking is done on the argument values.</p>

TASK SERVICES	
Schedule task	<p>void OPEX_sched_schedAt(OPEX_TCB *, year, month, day, hour, minute, second, tick)</p> <p><i>Purpose:</i> Reschedule a task to run at a specific date and time.</p> <p><i>First argument:</i> a pointer to the TCB to be rescheduled which is usually the current task's TCB.</p> <p><i>Second to nth arguments:</i> the desired date/time, declared as BYTE.</p> <p>Note: Minimal validity checking is done on the argument values.</p>
Find task by name	<p>OPEX_TCB *OPEX_sched_get_named(char *)</p> <p><i>Purpose:</i> Find the TCB for a named task..</p> <p><i>Returns:</i> TCB address or NULL if name not found.</p> <p><i>First argument:</i> a pointer to a RAM-resident, case-sensitive string which must match that of a task in the schedQ or flagwaitQ.</p> <p>Note: Some tasks may have been created as unnamed with a NULL name pointer.</p> <p>Note: The first matching name task is returned if there are two or more with the same name.</p>
Event wait	<p>void OPEX_sched_on_flag(flag *, mask)</p> <p><i>Purpose:</i> Suspend a task until an event occurs. An event is the writing, by another task, or an interrupt service routine, of the flag byte in RAM which is the first argument, subject to the bit mask given in the second argument. To wait for a single event, the mask contains a single bit set to 1.</p> <p>A task may pass a mask of 0 in which case the il be removed from the waiting queue is by another task with the quit or reschedule functions.</p> <p><i>First argument:</i> a pointer to a RAM resident BYTE with 8 possible event flag bits.</p> <p><i>Second argument:</i> A byte with with the desired event flag bit(s) set .</p> <p>When a task or an interrupt service routine <i>alters</i> (sets or clears) a bit matching the address and mask, the waiting task is scheduled to run immediately. The waiting task may repeat the wait based on the new state of the event bit(s) and which did change.</p> <p><i>Returns:</i> Does not return; it queues the task and jumps to the scheduler.</p> <p>While waiting for an event, the task is held in the flagwaitQ which is sorted as last in first out.</p> <p>Note: When rescheduled, previously waiting the task runs the function whose address is in the task's TCB, starting at the top of the function, not at the code just after OPEX_sched_on_flag(). See examples on state machines by using a switch() or by changing function pointers..</p>

TASK SERVICES	
Event trigger	<p>void OPEX_sched_change_flag(*flags, bitno, state)</p> <p><i>Purpose:</i> Trigger an event for which one or more tasks may be waiting having used OPEX_sched_on_flag().</p> <p><i>First argument:</i> a pointer to a RAM resident BYTE with 8 possible event flag bits.</p> <p><i>Second argument:</i> the bit number (0 to 7) of the event bit to be altered</p> <p><i>Third argument:</i> Non-zero if the event bit is to be set, else the event is cleared.</p> <p>This function will move all tasks waiting the matching events from to the schedQ marked to run immediately.</p> <p>Note: An interrupt service routine may call this function; the queue manipulation is ISR-safe.</p>

DATE / TIME

DATE / TIME SERVICES	
Timekeeping initialization	void OPEX_init_dateTime(void) Must be called before the hardware timer is enabled.
Put Current Date/time into a TCB	void OPEX_sched_now(OPEX_TCB *) <i>Purpose:</i> copy the current date and time into the TCB passed, normally but not necessarily the tasks own TCB. The TCB's date/time is the date/time at which the task will run, unless the task is waiting for an event. This can be used just prior to calling OPEX_sched_resched() to assure that the delay time will be relative to "now", to include tick.
Day of week calculation	int OPEX_DayOfWeek(Year, Month, Day) <i>Purpose:</i> Calculate the day of the week for a specified date. Zero is Sunday.
Compare two dates/times	int OPEX_compare_dt(DATE_TIME *t1, DATE_TIME *t2) <i>Purpose:</i> Compare two date/time values. The arguments are OPEX DATE_TIME structure pointers. <i>Returns:</i> -1 if t1 < t2; 0 if t1 == t2; +1 if t1 > t2
Format time to a string	void OPEX_format_time(char *buffer, DATE_TIME *t) <i>Purpose:</i> Place a text string formatted time into the buffer passed in the first argument, for the time in the second argument.
Format date to a string	void OPEX_format_date(char *buffer, DATE_TIME *t) <i>Purpose:</i> Place a text string formatted date into the buffer passed in the first argument, for the date in the second argument.
Format time and date to a string	void OPEX_format_date(char *buffer, DATE_TIME *t) <i>Purpose:</i> Place a text string formatted time and date into the buffer passed in the first argument, for the time and date in the second argument.
Add an offset to a date and time	void OPEX_date_add(DATE_TIME *, days, hours, minutes, seconds, ticks) <i>Purpose:</i> Add an offset to the date/time in a DATE_TIME structure. Can be used by a task instead of OPEX_sched_resched(). The OPEX_TCB pointer to the DATE_TIME structure in the TCB is used.
Place date/time values into a DATE_TIME structure	void OPEX_date_stuff(DATE_TIME *, year, month, day, hour, minute, second, tick) <i>Purpose:</i> Place the date and time into an OPEX structure of type DATE_TIME. An instance in RAM of this structure is used for time keeping. Also, all OPEX_TCB structures contain DATE_TIME for use in scheduling the task. Note:: No error checking is performed on the arguments. Note; If the global variable "struct DATE_TIME time" is used as the first argument, the OPEX clock/calendar will be set. In this case, these functions must also be called: OPEX_daylight_saving_adjust() OPEX_sched_time_changed()
Reschedule all tasks due to a time	void OPEX_sched_time_changed(void) <i>Purpose:</i> Cause the scheduler to change all tasks scheduled to run in the

OPEX v1.0

DATE / TIME SERVICES	
or date change	future to instead run immediately. All tasks may choose to reschedule themselves based on the new date and time setting. Tasks waiting on an event are not rescheduled.
Daylight Saving time adjustment update	void OPEX_daylight_saving_adjust() <i>Purpose:</i> adjust the daylight OPEX saving settings and future task due to a change of date and time.
Daylight Saving in effect (query)	int OPEX_is_daylight_saving(void) Returns non-zero if the current date/time is during daylight saving.
Daylight Saving start/end date/time	void OPEX_daylight_saving_changes_at(flag, DATE_TIME *t) Returns the date and time at which daylight saving changes. <i>First argument:</i> specifies if the start (1) or the end (0) of the daylight savings period is to be returned. <i>Second argument</i> is the structure to receive the result. Note: The start/end of daylight saving are constants. The choice of which constant to use is by the value of the GMT offset in the current date time, for North America and most of Europe. No other areas are supported with the constants in use now.

SERIAL OUTPUT

SERIAL OUTPUT SERVICES	
Serial I/O Buffer Initialization	void OPEX_com_init(void) Hardware initialization must be coded by the user in module "AVRdependent.c" In that C file, other hardware devices are initialized such as the timer used by OPEX for scheduling and time keeping.
Output a byte (non-blocking)	Int OPEX_putcNoBlock (BYTE) <i>Purpose:</i> Transmit an 8 bit byte (binary or ASCII). If the serial chip is busy transmitting prior data, the new data is buffered for delayed transmission. The OPEX Serial I/O interrupt handler empties this buffer to the serial port using routines in the module "AVRdependent.c" Note: If the output buffer is full, this function does not buffer the data and returns non-zero. If the data can be stored in the output buffer, or if the buffer was empty, the function returns zero.
Output a byte (blocking)	void OPEX_putc(BYTE) <i>Purpose:</i> Place a byte which is binary or ASCII data into the serial output buffer. The OPEX Serial I/O interrupt handler empties this buffer to the serial port using routines in the module "AVRdependent.c" Note: If the output buffer is full, this function loops, calling OPEX_putcNoBlock() until there is room for the new data. This may undesirably affect task timing. To preclude this blocking , a task may (1) call OPEX_txbuf_unused() to check for buffer not full, or (2) call OPEX_putcNoBlock() and check the return value; if non-zero, the task may reschedule itself to pause and try later.
Output a new line character	void OPEX_nl(void) <i>Purpose:</i> A byte containing 0x0D which is 'r' is sent. Uses OPEX_putc() See this note
Output a string stored in RAM	void OPEX_puts(char *p) <i>Purpose:</i> Copy a null terminated ASCII string stored in RAM to the serial output buffer, using OPEX_putc() . A string literal "something" is by default in RAM, unless a compiler directive is used to place it in program memory, in which case OPEX_puts_P() must be used. Uses OPEX_putc() See this note
Output a string stored in RAM with a new line	void OPEX_putline(char *p) <i>Purpose:</i> Same as OPEX_puts() , followed by a call to OPEX_nl() Uses OPEX_putc() See this note
Output a string in stored program memory	void OPEX_puts_P(char *p) <i>Purpose:</i> Copy a null terminated ASCII string stored in program memory to the serial output buffer, using OPEX_putc() . The string must be a constant declared as program-memory-resident to the compiler, e.g., with GCC's PSTR("something") macro.

SERIAL OUTPUT SERVICES	
	<p>Uses OPEX_putc() See this note</p>
Output a string stored in program memory with a new line	<p>void OPEX_putline_P(char *p) <i>Purpose:</i> Same as OPEX_puts_P(), followed by a call to OPEX_ni() Uses OPEX_putc() See this note</p>
Get buffer unused count	<p>int OPEX_txbuf_unused(void) <i>Purpose:</i> Returns the size of the empty portion of the serial output buffer. If the size is zero, the next serial output OPEX function call <i>may</i> block (loop). <i>To avoid blocking, a task may elect to reschedule its output for future time if this function returns zero or less than the desired size.</i></p>
Get buffer used count	<p>int OPEX_txbuf_used(void) <i>Purpose:</i> Returns the number of bytes <i>used</i> in the serial output buffer</p>

SERIAL INPUT

SERIAL INPUT SERVICES	
Serial I/O Buffer Initialization	void OPEX_com_init(void) Hardware initialization must be coded by the user in module "AVRdependent.c" In that C file, other hardware devices are initialized such as the timer used by OPEX for scheduling and time keeping.
Check for input data available and peek first/last	Int OPEX_com_peek(unsigned char *c, BYTE last) <i>Purpose:</i> Returns the number of bytes now in the serial input buffer. If the buffer is not empty, the first argument, a byte pointer, is used to store a byte from the input buffer. The second argument specifies whether the byte retrieved from the buffer is the first (zero) or last (non-zero) in the buffer.
Get a byte from the input buffer if available (non-blocking)	int OPEX_getc(unsigned char *p) <i>Purpose:</i> Returns zero if no data is available in the serial input buffer. If data is available, returns non-zero, removes the data from the buffer, and stores it using the argument, a pointer. Thus, this function never blocks or loops waiting for data.
Get a string from the serial input, with a timeout (optionally non-blocking)	int OPEX_gets(char *p1, int maxchars, BYTE timeout_secs) <i>Purpose:</i> Reads a string terminated by an ASCII CR (0x0D). Returns the number of characters stored in the buffer pointed to by argument one, up to the limit set by argument two, which should allow room for the null byte terminator. If argument three is non-zero, the function loops up to n seconds while expecting a CR. If argument three is zero, the program blocks (loops) indefinitely waiting for the CR. The CR is stored in the buffer, followed by a NULL. The returned string may not contain a CR if <i>maxchars</i> is reached or a <i>timeout</i> occurs. Incoming ASCII 0x0A which is '\n', is ignored. Note: To avoid blocking, a task may set the third argument to zero or it may use OPEX_com_peek() or OPEX_getc() , above which do not block.

MONITORING AND DEBUGGING

MONITORING AND DEBUGGING SERVICES	
Enable or Disable Monitoring	OPEX_set_monitoring(BYTE) <i>Purpose:</i> Enable or disable formatting and output of monitoring functions listed below. If disabled, the functions below return with minimal computation. The argument, if non-zero, enables.
Redirect monitoring output text	void * OPEX_sched_monitor_redirect(void (*fp) (char)) <i>Purpose:</i> Redirect text output from the monitoring functions which are listed below. The argument is a pointer to a function that the routines below will call, passing a character. If the argument is NULL, the redirect is disabled and the output reverts to the default. By default, the monitoring output goes to the OPEX serial output routine. <i>Returns:</i> Previous value of the redirect function pointer. Note: Redirecting to a function which merely returns is, of course, a way to discard monitoring output, though the CPU load to compute the text remains. One could redirect output to a user supplied I/O routine.
Get Status of Monitoring enable/disable	int OPEX_monitoring(void) <i>Purpose:</i> Get the current Enable/Disable monitoring setting. Returns non-zero if enabled.
Output a Report on a task	void OPEX_task_report(char *p, OPEX_TCB *f) <i>Purpose:</i> Can be used by a task to report that it has run. Format and output a formatted report on the task who's TCB is passed as the second argument. Includes the current date/time. The TCB passed can be any task's TCB, but it may be confusing if it is not the TCB of the calling task.
Output One TCB's status	void OPEX_sched_showQitem(OPEX_TCB *, char *, char) <i>Purpose:</i> Format the status of a TCB to text. The TCB may be a member of the linked list (queue) who's head pointer is schedQ (the time sorted queue) or flagwaitQ (tasks waiting on flags). Argument 2 is a pointer to the buffer to receive the text. Argument 3, if non-zero, causes the TCB's flags to be included in the formatted output. Note: A task may use this function to "log" when it runs or when it runs with certain conditions as might be noted in the TCB's <i>state</i> or <i>pinfo</i> fields.
Output All TCB's status	void OPEX_sched_showQ(char *, BYTE) <i>Purpose:</i> Format the status of every TCB in the OPEX linked lists (queues) for tasks waiting on a date/time and those waiting on flags. Arguments 1 and 2 are as in OPEX_sched_showQitem() arguments 2 and 3. Beware: When many tasks are active, the output may be lengthy and cause blocking on the serial output stream. To avoid this, use OPEX_sched_showQitem() in a list-walking loop with code to delay if the output buffer is full. See this note

MONITORING AND DEBUGGING SERVICES

Output memory statistics

void OPEX_sched_show_mem(char *, unsigned int)

Purpose: Format the memory statistics to text. Argument 1 is a pointer to the buffer to receive the text. Argument 2 is the size of the RAM for this microprocessor.

OPEX Data Structures

OPEX DATE_TIME STRUCTURE

```
typedef struct time_date_node { // must be ordered highest to lowest
    unsigned char  year;          // 0..99 this century
    unsigned char  month;
    unsigned char  day;
    unsigned char  hour;
    unsigned char  minute;
    unsigned char  second;
    unsigned char  tick;
    unsigned char  weekday;       // 0..6 not used in comparisons
} DATE_TIME ;
```

GMT offset is stored separately.

OPEX TASK CONTROL BLOCK (OPEX_TCB) STRUCTURE

```
typedef struct func_state_node {
    struct func_state_node *next; // linked list
    DATE_TIME when;              // DATE AND TIME when this task is to run
    BYTE state;                  // user defined
    void *pinfo;                 // user defined any 16 bits
    size_t *malloc_list;        // list of malloc'd memory
    char *name;                  // string name (RAM data)
    void (*funcp) ( struct func_state_node *me); // function to run
    BYTE *flags;                 // pointer to 8 flag bits
    BYTE flagmask;               // flag mask for flagwait
} OPEX_TCB ;
```

OPEX QUEUES

The global *schedQ* is a pointer to the first OPEX_TCB in a linked list of tasks to run at given times. The list is kept ordered by date and time, youngest first.

The global *flagwaitQ* is a pointer to the first OPEX_TCB in a linked list of tasks waiting on a flag change event. The list is ordered chronologically. If *flagwaitQ* is NULL (0), the queue is empty.

When a task is removed from the *flagwaitQ* due to a flag event, such as in an interrupt routine, OPEX places it in a linked list of pending tasks. Each TCB in the pending list is inserted into the front of the *schedQ* before any tasks are run.