

9 PaulOS – a Co-operative RTOS

The PaulOS (PAUL's Operating System) co-operative RTOS is described here. This is the 'flagship' RTOS which we regularly use during the year with our students. It is heavily used also for their final year theses and it has been regularly refined to reflect the changes and upgrading requested by the students as they became more and more familiar with the performance and limitations of this co-operative RTOS. In this RTOS, each task is free to run for as long as it wishes. The task itself can control when to give up the processor time to allow other tasks to run.

The original idea for this RTOS came from the book "C and the 8051 – Building Efficient Applications – Volume II" by Thomas W. Schultz.¹ This RTOS is a direct adaptation of my PaulOS assembly language program, re-written in C so as to make it more versatile and more easily portable to other micro-controllers. In fact it was even successfully ported to the Intel 8086 microprocessor and an 8086 version with an example is also given in the Appendix. The main task of translating it from assembly to C was undertaken years ago as a final year engineering degree thesis [20] (Blaut 2004), then a student under my supervision. It was further developed and improved throughout the years by myself, thanks also to input and suggestions from other students taking my study-units during their degree program, into the version shown here. I consider this RTOS as providing a good basis to the study of a real-time operating system for the 8051.



www.sylvania.com

We do not reinvent the wheel we reinvent light.

Fascinating lighting offers an infinite spectrum of possibilities: Innovative technologies and new markets provide both opportunities and challenges. An environment in which your expertise is in high demand. Enjoy the supportive working atmosphere within our global group and benefit from international career paths. Implement sustainable ideas in close cooperation with other specialists and contribute to influencing our future. Come and join us in reinventing light every day.

Light is OSRAM

OSRAM SYLVANIA

Most of the commands are exactly the same, (with the additional OS_ prefix) as explained in PaulOS.a51 RTOS assembly language version, also found in the appendix. The settings regarding the number of tasks and stack size and location can be set in the parameters file, which is also listed at the end of this chapter.

There are some very immediate advantages in using C to write the RTOS. Parameters can be easily changed from char to integer or long types and the routines would automatically reflect the changes when they are compiled. An example here would be the 'wait for timeout' OS_WAITT(parameter) command where in the A51 version, the parameter was of type integer (0-65535). In the A51 version, if we had to change the parameter to long in order to be able to accommodate longer wait periods, we would have to re-write the routines so as to increment or decrement double words (32-bit) rather than words (16-bit). In the C version this could be done fairly easily simply by changing the type declarations. The compiler would do the rest.

Naturally there are some memory space and speed penalties to pay for this versatility. However the improvements are more than worth the penalties, especially as far as student understanding of the RTOS is concerned. In the next paragraph we now list once again the RTOS commands, including the improvements, mainly achieved with the use of MACROS which are listed in section 9.3.14. The full source program can be found in appendix D.

9.1 Description of the RTOS Operation

The PaulOS RTOS is a co-operative RTOS and hence, as explained earlier in the RTOS chapter, each task has to take the initiative to give up its own time so as to allow other tasks to run. It has to be kept in mind that this OS is running on an 8051-based micro-controller which can only run one program at a time and hence this task swapping RTOS only gives the impression of having tasks running simultaneously. In actual fact we can only have one task actually running, and at the time that the RTOS is doing its own checks, no tasks at all would be running. This time ideally should be kept as short as possible.

The operation of the RTOS is as follows:

Each task, when created, would have its own memory area in external memory where there would be stored all the registers (R's, A, B, DPTR, PSW), stack area (including the return address of the task or function). Once a change of task is required, the RTOS would take care to swap the relevant registers and stack areas so that the micro-controller would have the correct data for the new task in its own internal RAM.

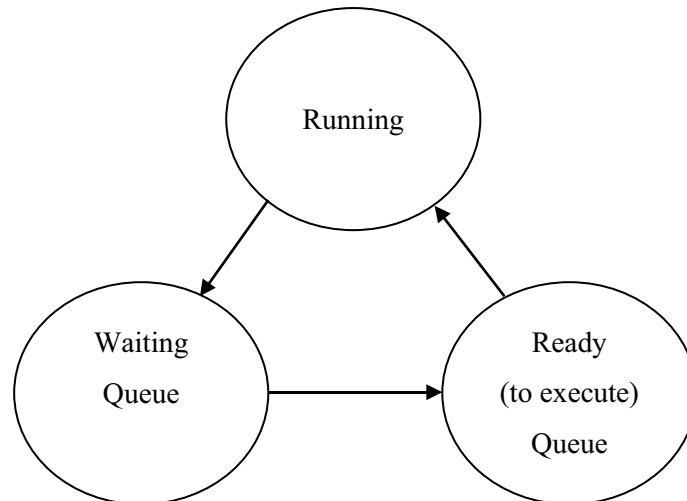


Figure 9-1 RTOS Task states diagram

The RTOS tick-timer can be chosen by the user who can select from the different timers available on the controller. Once set, at every timer overflow, an interrupt call is made to the main RTOS tick timer interrupt service routine. This is the most important routine in the program since at every interrupt the RTOS has to check the status of all the tasks so as to be able to decide whether a task can be moved from the Waiting queue to the Ready queue (see Figure 9-1) or a task swap if the main() was running is required. The RTOS achieves this by counting down the parameter variables holding the individual waiting time required for those tasks in the waiting queue. When anyone of these timeout parameters reaches 0, it means that the time to move on has arrived. Once again, being a co-operative RTOS, the scheduler cannot swap tasks on its own accord. Only the main() code can be forced to give up its time, so that if at any time whilst the main() code is running, there is a task which moves into the Ready queue, then that task takes over.

On the other hand, when one of the OS commands which forces a task change is encountered in a task, then it is only at that instance that a task swap is implemented. The currently running task is marked as being in the Waiting queue and the first task in the Ready queue takes over, with the stack and registers being conveniently swapped.

The idea behind the PaulOS RTOS is that any task (a function or a routine in a program) can be in any ONE of three states, Running, Waiting (for some event or time delay) or Ready (to execute) state.

RUNNING

A task can be RUNNING, (obviously in the single 8051 environment, there can only be one task which is actually in the running state). If there are no tasks which are ready to execute, then the RTOS will set the main() as the running task. This will be interrupted at any time, as soon as a task becomes ready to run.

WAITING

A task can be in the WAITING (sometimes also referred to as SLEEPING) queue. Here a task could be waiting for any one of the following time delays or events to occur:

- a specified amount of time delay, selected by the user with OS_WAITT command. OS_DEFER command is actually an OS_WAITT(2) – wait for 2 ticks.
- a specified amount of time delay, selected by the user with OS_PERIODIC command. The actual task is placed in the waiting queue when the OS_WAITP (wait for periodic interval) is encountered.
- a specified interrupt to occur within a specified time, selected by the user with the OS_WAITI command.
- a signal from some other task within a specified timeout, selected by the user with the OS_WAITS(timeout) command.
- a signal from some other task indefinitely, selected by the user with the OS_WAITS(0) command.
- a never-ending waiting period. A task could be waiting here indefinitely, effectively behaving as if the task did not exist. This is specified by the OS_KILL_IT command.



360°
thinking.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.



READY

It can also be in the READY QUEUE, waiting for its turn to execute. This can be visualised in Figure 9-1 which shows how the tasks can move from one state to another. The RTOS, when permitted to do so, will select the top task from this queue to execute instead of the currently running task, which would then be placed in the waiting queue.

The RTOS itself always resides in the background, and comes into play:

- At every RTOS TIMER interrupt (usually when Timer 2 or Timer 0 overflows, say every one millisecond) so as to update the waiting time left for any tasks.
- At any other interrupt from other timers or external inputs so as to check whether it needs to move to the ready queue any tasks which were waiting for such events or interrupts.
- Whenever an RTOS system command is issued by the main program or tasks, to perform that system command.

The RTOS which is effectively supervising and scheduling all the other tasks, then has to make a decision whether it has to pause the current task and resume a new one or whether it can let the current task run on. There could be various reasons for changing tasks, as explained further on, but in order to do this task swap smoothly, the RTOS has to save all the environment of the presently running task and substitute it with the environment of the next task which is about to run. This is accomplished by saving all the BANK 0 registers, the ACC, B, PSW, and DPTR registers. The STACK too has to be saved since the task might have pushed some data on the stack (apart from the address at the point that the task was interrupted, where it has to return to after the interrupt). This is the crux of the PaulOS RTOS.

9.2 PaulOS.C System Commands

We now list and explain all the PaulOS RTOS system commands. These are first listed or grouped according to whether or not they take any parameters. The list is then repeated, this time sorted according to whether the command causes a task swap or not.

The following RTOS system calls do not receive any parameters :

- OS_DEFER (void); // Stops current task and passes control to next task in queue
- OS_KILL_IT (void); // Kills a task - sets it waiting forever
- OS_RUNNING_TASK_ID(void); // Returns the task number of the currently executing task
- OS_SCHECK (void); // Checks if running task's signal bit is set, returns a bit value
// of 1 if signal is already present.
- OS_WAITP (void); // Waits for end of task's periodic interval, set by
// the OS_PERIODIC command.

The following RTOS system calls do receive parameters:

- OS_CREATE_TASK (uchar tasknum, uint taskadd); // Creates a task
- OS_INIT_RTOS (uchar iemask); // Initialises all RTOS variables
- OS_PERIODIC (uint ticks); // Tasks run periodically every number of ticks
- OS_RESUME_TASK (uchar tasknum); // Resumes a task which was previously KILLED
- OS_RTOS_GO (uchar prior); // Starts the RTOS with priorities if required
- OS_SIGNAL_TASK (uchar tasknum); // Signals a task
- OS_WAITI (uchar intnum); // Waits for an event (interrupt) to occur
- OS_WAITS (uint ticks); // Waits for a signal within a number of ticks
- OS_WAITT (uint ticks); // Waits for a timeout defined by number of ticks

The list of commands can also be grouped as those which cause a change of task, might cause a change of task and those which do not cause a task swap.

The following RTOS system calls force a task change after executing this command:

- OS_DEFER (void); // Stops current task and passes control to next task in queue
- OS_KILL_IT (void); // Kills a task – sets it waiting forever
- OS_WAITI (uchar intnum); // Waits for an event (interrupt) to occur
- OS_WAITT (uint ticks); // Waits for a timeout defined by number of ticks
- OS_WAITP (void); // Waits for the end of the task's periodic interval

The following RTOS system calls might force a task change after executing this command:

- OS_WAITS (uint ticks); // Waits for a signal within a number of ticks

If the signal is already present when the command is issued, then no task swap is made, otherwise a task change is performed.

The following RTOS system calls do not force a task change, and the task using any of these commands would continue to run after executing the command:

- OS_CREATE_TASK (uchar tasknum, uint taskadd); // Creates a task
- OS_INIT_RTOS (uchar iemask); // Initialises all RTOS variables
- OS_PERIODIC (uint ticks); // Tasks run periodically every number of ticks
- OS_RESUME_TASK (uchar tasknum); // Resumes a task which was previously KILLED

- OS_RTOS_GO (uchar prior); // Starts the RTOS with priorities if required
- OS_RUNNING_TASK_ID(void); // Returns the task number of the currently running task
- OS_SCHECK (void); // Checks if running task's signal bit is set
- OS_SIGNAL_TASK (uchar tasknum); // Signals a task

9.3 Descriptions of the commands

The C version of the RTOS provides some variations and additional commands which were implemented after having used the A51 program for a while. Some of the additions were only implemented in the C version although they can be easily added in the assembly version as well. The detailed description of the commands now follows, which would completely describe the RTOS. The complete PaulOS RTOS source program can be found in the Appendix D and examples are given at the end of this chapter which should make it easier to understand.


9.3.1 OS_INIT_RTOS(IEMASK)

This system command must be the **first** command to be issued in the main program in order to initialise the RTOS variables. It is called from the main program and takes the interrupt enable mask (IEMASK) as a parameter. An example of the syntax used for this command is:

```
OS_INIT_RTOS(0x30);
```

SIMPLY CLEVER

ŠKODA



We will turn your CV into an opportunity of a lifetime



Do you like cars? Would you like to be a part of a successful brand? We will appreciate and reward both your enthusiasm and talent. Send us your CV. You will be surprised where it can take you.

Send us your CV on www.employerforlife.com

which would imply that the application program intends to use the Timer 2 interrupt (IEMASK=20H) for the RTOS as well as the Serial Interrupt (IEMASK=10H). Hence the 0x30 parameter in the command.

Interrupt		IE MASK		Notes
No:	Name	Binary	Hex	
0	External Int 0	00000001	01	
1	Timer Counter 0	00000010	02	Default RTOS timer for 8051
2	External Int 1	00000100	04	
3	Timer Counter 1	00001000	08	
4	Serial Port	00010000	10	
5	Timer 2 (8032 only)	00100000	20	Default RTOS for 8032

Table 9-1 IEMASK Parameter (PaulOS)

The correct mask for the RTOS timer (defined in the file parameters.h) is always added (or ORed) by the RTOS automatically to any other mask, even if one forgets to enable it in the IEMASK parameter. The interrupts which are valid are shown in Table 9-1. This implies that in order to change the tick timer (that is the interrupt number for the RTOS) we have to change the TICK_TIMER parameter in the parameters.h file.

This system command performs the following:

- Clears the external memory area which is going to be used to store the stack of each task.
- Sets up the IE register (location A8H in the SFR area).
- Selects edge triggering on the external interrupts. This can be amended if a different triggering is required by changing directly the default initialisation in the RTOS source code listing found in Appendix D or by re-setting the correct triggering mode after having initialised the RTOS so as to override the default value. This is done by setting the correct bit value for IT0 and IT1 residing in the TCON SFR as already stated in section 2.10.16.
- Loads the Ready Queue with the main idle task number, so that initially only the main task will execute.
- Initialises all tasks as being not waiting for a timeout.
- Sets up the Stack Pointer (SP) variable of each task to point to the correct location in the stack area of the particular task. The stack pointer, initially, is made to point to an offset of 14 bytes above the base of the stack [(MAIN_STACK - 1) + NOOFPUSHES + 2] since NOOFPUSHES in this case is 13. The first 13 locations would initially all contain a zero. This is done so as to ensure that when the first RET instruction is executed after transferring the stack from external RAM on to the 8032 RAM, the SP would be pointing correctly to the address of the task to be started. This is seen in the QSHFT routine, where before the last RET instruction, there is the Pop_Bank0_Reg macro which effectively pops 13 registers. The RET instruction would then read the correct address to jump to from the next 2 locations.

9.3.2 OS_CREATE_TASK(Task No;, Task Name)

This system command is used in the main program for each task to be created. It takes two parameters, namely the task number (the first task is normally numbered as task 0), and the task address, which in the C environment, would simply be the name of the procedure or function. An example of the syntax used for this command is:

```
OS_CREATE_TASK(0, MotorOn);
```

This would create a task, numbered 0 which would refer to the MorotOn() procedure or function.

This system command performs the following:

- Places the task number in the next available location in Ready Queue, meaning that this task is ready to execute. The location pointer in Ready Queue is referred to as READYQTOP in the program, and is incremented every time this command is issued.
- Loads the address of the start of the task at the bottom of the stack area in external ram allocated to this task. The SP for this task would have been already saved, by the INIT_RTOS command, pointing to an offset 13 bytes above this, to compensate for the pops.

9.3.3 OS_RTOS_GO(Priority)

This system command is used only ONCE in the main program, when the RTOS would be required to start supervising the processes. It takes one Priority bit parameter.

The Priority bit parameter (0 or 1) if set to 1, implies that those tasks placed in the Ready Queue (meaning ready to execute), would be sorted in descending order before the RTOS selects the next task to run. A task number of 0 is taken to mean by the RTOS as the **highest** priority task, and would obviously be given preference during the sorting. The main() task or function is automatically given the highest task number (thus meaning the lowest priority) by the RTOS, so as all the other tasks would be able to interrupt it.

An example of the syntax used for this command is:

```
OS_RTOS_GO(1);
```

This would start the RTOS ticking with priority enabled. The tick time interval is determined by the parameter TICKTIME set in the parameters header file (say 1ms, 5ms or 10ms). This value would then become the basic reference unit for other system commands which use any timeout parameter.

The RTOS would also be required to execute “ready-tasks” sorting prior to any task change, since the priority parameter was set to 1.

Assuming Timer 2 is being used to generate the ticktime this system command performs the following:

- Loads the variable DELAY (LO and HI bytes), with the number of BASIC_TICKS required to obtain the required ticktime delay.
- Sets the PRIORITY bit according to the priority parameter supplied.
- Loads RCAP2H and RCAP2L, the Timer 2 registers, with the required count in order to obtain the required delay between Timer 2 overflow interrupts. The value used depends on the crystal frequency used on the board. The clock registers count up at one twelfth the clock frequency, and using a clock frequency of 11.0592 MHz, each count would involve a time delay of $12/11.0592 \mu\text{s}$ or $1.08507 \mu\text{s}$. Therefore to get a delay of 1ms (1000 μs), $1000/1.08507$ or 921.6 counts would be needed. We would use integer 921 to get this delay, hence the reload registers (RCAP2H,RCAP2L) would be loaded with $65536 - 921$ since the timers count up till they overflow.
- Stores the reference time signal parameter in GOPARAM and TICKCOUNT.
- Starts Timer 2 in 16-bit auto-reload mode.
- Enables interrupts.
- Sets TF2, which is the Timer 2 overflow interrupt flag, thus causing the 1st interrupt immediately.

9.3.4 OS_RUNNING_TASK_ID()

This system command is used by a task to get the number of the task itself. It returns an unsigned character (1 byte) value and the same task continues to run after executing this system command.

An example of the syntax used for this command is:

```
X = OS_RUNNING_TASK_ID(); /* where X would be an unsigned character */
```

9.3.5 OS_SCHEK()

This system command is used by a task to test whether there was any signal sent to it by some other task.

- It returns a bit value of:
 - 0 if Signal is not present
 - 1 if Signal is present
- If the signal was present, the signal flag (bit) is also cleared before returning to the calling task. The same task continues to run, irrespective of the returned value.

An example of the syntax used for this command is:

```
X = OS_SCHEK(); /* where X would be a bit-type variable */
```

or one may use it as in the following example to test the presence of the signal bit:

```
if (OS_SCHEK() == 1)
{
/* do these instructions if a signal was present */
}
```

9.3.6 OS_SIGNAL_TASK(Task No:)

This system command is used by a task to send a signal to another task. If the other task was already waiting for a signal, then the other task is placed in the Ready Queue and its waiting for signal flag is cleared. The task issuing the OS_SIGNAL_TASK command continues to run, irrespective of whether the called task was waiting or not waiting for the signal. If we need to halt the task after the OS_SIGNAL_TASK command to give way to other tasks, we must use the OS_DEFER() system command after the OS_SIGNAL_TASK command.



I joined MITAS because
I wanted **real responsibility**

The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com

Month 16
I was a construction
supervisor in
the North Sea
advising and
helping foremen
solve problems

Real work
International opportunities
Three work placements



This system command performs the following:

- It first checks whether the called task was already waiting for a signal.
- If the called (signaled) task was not waiting, it sets its waiting for signal (SIGW) flag and exits to continue the same task.
- If the signaled task was already waiting, it places the called task in the Ready Queue and it clears both the waiting for signal (SIGW) and the signal present (SIGS) flags.
- It also sets a flag (TINQFLAG) to indicate that a new task has been placed in the Ready Queue. This flag is used by the RTOS_TIMER_INT routine (every half a millisecond) in order to be able to decide whether there has to be a task change. It then exits the routine to continue the same task.

An example of the syntax used for this command is:

```
OS_SIGNAL_TASK(1);      // send a signal to task number 1
OS_DEFER();             // give cpu time to other tasks, if necessary
```

9.3.7 OS_PERIODIC (uint ticks)

This command initialises the task to repeat periodically, every certain number of ticks given as a parameter in the command. It is used at the beginning of a task, OUTSIDE of the endless loop, as shown in the next sub-section 9.3.8. An example of its usage is also given in that same sub-section.

We now deal with the commands that do perform a voluntary (co-operative) change of task:

9.3.8 OS_WAITP (void)

This command sets the task waiting for the preset periodic interval (set previously by the OS_PERIODIC(ticks) command). The task goes into a waiting state and the next ready task takes over.

If the interval has already passed when this command is executed, then the task would continue to execute. This is not normally the case, and only happens when there is a programming logic or algorithm mistake, since it would generally mean that the task is actually taking longer to execute than the requested periodic interval between executions.

It performs the following:

- Saves task environment in preparation for the expected task swap.
- If the periodic interval has not yet passed, as is generally the case, it sets the periodic interval flag to indicate that it is waiting for the periodic interval and issues a voluntary task change.

- If however the periodic interval has already elapsed (this is usually due to bad programming, in cases where the code of the task itself takes a longer time to execute than the required periodic interval), then it clears the periodic interval flag and exits.

Such a command is used in a task, in conjunction with the OS_PERIODIC() command and an example of its usage is shown below:

```
OS_PERIODIC(50);           // declare task as wishing to execute every 50 ticks
while(1)                   // repeat forever
{
....                       // code to be executed every 50 ticks
....                       // which should not take longer than
....                       // 50 ticks to execute.
OS_WAITP();                // wait for the periodic interval to pass
}
```

9.3.9 OS_WAITI(Interrupt No:)

This system command is called by a task to sleep and wait for an interrupt to occur. Another task, next in line in the Ready Queue would then take over. If the interrupt never occurs, then the task will effectively sleep for ever. This is one way of writing Interrupt Service Routines under PaulOS RTOS control. ISRs can also be written in such a way as to run independently, as describe in section 9.3.15.

If required, this command can be modified to allow another timeout parameter to be passed, so that if the interrupt does not arrive within the specified timeout, the task would resume. A timeout of 0 would still leave the task waiting for the interrupt forever. The modification required to the RTOS source listing would be similar to the OS_WAITTS command, and the operation would then be as explained further down in sub-section 9.3.10.

This system command performs the following:

- It sets the bit which corresponds to the interrupt number passed on as a parameter.
- It then calls the QSHFT routine in order to start the task next in line.

An example of the syntax used for this command is:

```
OS_WAITI(0);              // wait for an interrupt from external int 0
```

The task would then go into the sleep or waiting mode and a new task would take over.

9.3.10 OS_WAITS(Timeout)

This system command is called by a task to sleep and wait for a signal to arrive from some other task. If the signal is already present (previously set or signaled by some other task), then the signal is simply cleared and the task continues on. If the signal does not arrive within the specified timeout period, the task resumes just the same. However, a timeout number of 0 would imply that the task has to keep on waiting for a signal indefinitely. If the signal does not arrive, then the task never resumes to run and effectively the task is killed.

This system command performs the following:

- It first checks whether the signal is already present.
- If the signal is present, then it clears the signal flag, exits and continues running.
- If the signal is not present, then:
 - It sets its own waiting for signal (SIGW) flag.
 - It also sets the waiting for timeout variable according to the supplied parameter.
 - It then jumps to the QSHFT routine in order to start the task next in line.

ie business school

#1 EUROPEAN BUSINESS SCHOOL
FINANCIAL TIMES 2013

#gobeyond

MASTER IN MANAGEMENT

Because achieving your dreams is your greatest challenge. IE Business School's Master in Management taught in English, Spanish or bilingually, trains young high performance professionals at the beginning of their career through an innovative and stimulating program that will help them reach their full potential.

- Choose your area of specialization.
- Customize your master through the different options offered.
- Global Immersion Weeks in locations such as London, Silicon Valley or Shanghai.

Because you change, we change with you.

www.ie.edu/master-management | mim.admissions@ie.edu | f t in YouTube



An example of the syntax used for this command is:

```
OS_WAITS(50);  
// wait for a signal within 50 units or ticks, the value of the unit depends on  
// the TICKTIME parameter used.
```

If for example, the TICKTIME was set to 10 milliseconds in the header file, an OS_WAITS(50) would then imply waiting for a signal to arrive within 500 milliseconds.

or you can use:

```
OS_WAITS(0); // this would wait for a signal for ever
```

In both examples, if the signal is not already present, the task would then go into the sleep or waiting mode and a new task would take over.

9.3.11 OS_WAITT(Timeout)

This system command is called by a task to sleep and wait for a specified timeout period. The timeout period is in units whose value depends on the TICKTIME parameter used. Valid values for the timeout period are in the range 1 to 65535. A value of 0 is reserved for the OS_KILL_IT command, meaning permanent sleep, and therefore is not allowed for this command. The OS_WAITT system command therefore performs the required check on the parameter before accepting the value. If by mistake a value of 0 is given as a timeout parameter, then it is automatically changed to a 1. Once the timeout period passes, the task which had issued this command, would be moved from the waiting to the ready queue.

This system command performs the following:

- If the parameter is 0, then set it to 1, to avoid permanent sleep.
- Save the correct parameter in its correct place in the TTS table.
- Jump to the QSHFT routine in order to start the task next in line.

An example of the syntax used for this command is:

```
OS_WAITT(60);  
// wait for a signal for 60 units, the value of the unit depends on  
// the TICKTIME parameter used.
```

If for example, the command TICKTIME was set to 10, the reference unit would be 10 milliseconds, and OS_WAITT(60) would then imply waiting or sleeping for 600 milliseconds. The task would then go into the sleep or waiting mode for 600ms and a new task would take over. After 600ms it would move to the ready queue.

9.3.12 OS_KILL_IT()

This system command is used by a task in order to stop or terminate the task. As explained earlier in OS_WAITT, this is simply the command OS_WAITT with an allowed timeout of 0. The task is then placed permanently waiting and never resumes execution.

This system command performs the following:

- First it clears any waiting for signal or waiting for interrupt flags, so that that task would definitely never restart.
- Then it sets its timeout period in the TTS table to 0, which is the magic number the RTOS uses to define any non-timing task.
- Then it sets the INTVLRD and INTVLCNT to 0, again implying that it is not a periodic task.
- Finally it jumps to the QSHFT routine in order to start the task next in line.

An example of the syntax used for this command is:

```
OS_KILL_IT();  
/* the task simply stops to execute and a new task would take over.*/
```

9.3.13 OS_DEFER()

This system command is used by a task in order to hand over processor time to another task. The task is simply placed at the end of the Ready Queue, while a new task resumes execution.

This system command performs the following:

- It sets its timeout period in the TTS table to 0, which is the magic number the RTOS uses to describe any non-timing task.
- It places the task in the Ready Queue, by simply placing the task number in the next available location in Ready Queue area.
- It then flows on to the QSHFT routine in order to start the task next in line.

An example of the syntax used for this command is:

```
OS_DEFER();  
  
/* the task simply stops execution and is placed in Ready Queue.*/  
  
/* A new task would then take over. */
```

9.3.14 Enhanced event-waiting and other add-on MACROS

These macros (#define statements) perform the same functions of the OS_WAITT, OS_WAITS and OS_PERIODIC calls but rather than ticks they accept absolute time values as parameters in terms of minutes, seconds and millisecs. This difference is denoted by the _A suffix (the A standing for Absolute) – eg. OS_WAITT_A(0,0,300) would cause a task to wait for 300ms and is the absolute-time version of OS_WAITT(x), where x would have to be calculated to give the required number of ticks equivalent to a 300ms delay.

Range of values (65535 TICKTIMES) accepted is listed below:

Using a minimum TICKTIME of 1ms :

Range from 1ms to 1m, 5s, 535ms in steps of 1ms.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"
Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download

Using a recommended TICKTIME of 10ms:

Range from 10ms to 10m, 55s, 350ms in steps of 10ms.

Using a maximum TICKTIME of 50 ms:

Range from 50ms–54m, 36s, 750ms in steps of 50ms

If the conversion from absolute time to ticks results in 0 (all parameters being 0 or overflow) this result is only accepted by OS_WAITS() by virtue of how the OS_WAITT(), OS_WAITS() and OS_PERIODIC() calls were written. In the case of the OS_WAITT() and OS_PERIODIC() calls the tick count would automatically be changed to 1 meaning an interval of 1 ticktime.

```
OS_WAITT_A(M,S,ms)    // Absolute OS_WAITT for minutes, seconds and milliseconds
OS_WAITS_A(M,S,ms)   // Absolute OS_WAITS for minutes, seconds and milliseconds
OS_PERIODIC_A(M,S,ms) // Absolute OS_PERIODIC for minutes, seconds and milliseconds

OS_PAUSE_RTOS()      // Disable the RTOS, used in a stand-alone ISR
OS_RESUME_RTOS()     // Re-enable the RTOS, used in a stand-alone ISR

OS_CPU_IDLE()        // Sets the µC in idle mode in PCON SFR (section 1.8.10).
                     // This is usually used in the main program endless loop after
                     // initialising and starting the RTOS.

OS_CPU_DOWN()        // Sets the µC in power-down mode in PCON SFR (section 1.8.10).
```

9.3.15 Stand-alone Interrupt Service Routines

In the C version of the RTOS, a simple method of having one or more stand-alone interrupt service routine (ISR) which would run whenever some interrupt is generated has been included.

All we have to do is to set to '1' the corresponding interrupt in the PaulOS.H file. For example if we intend to have an ISR running under the EXT 0 interrupt (and not under RTOS control), then we have to make sure to set to one the corresponding #define statement in PaulOS.H file. In some examples or listings shown in this book, these STAND_ALONE_ISR parameters were moved to the parameters.h header file so as to have all parameters which can be changed by the user in one file, but the effect is the same.

```
#define STAND_ALONE_ISR_00 1 // EXT0 – set to 1 if using this interrupt as a stand alone ISR
```

Then in the ISR itself we should also include the commands `OS_PAUSE_RTOS()` when starting the ISR and then `OS_RESUME_RTOS()` in order to resume the RTOS before exiting the ISR. This would ensure that the RTOS does not interfere with the stand-alone ISR.

It is best to use register banks 2 or 3 for these ISRs.

Example of a stand-alone ISR, interrupting the RTOS and executing immediately when the interrupt occurs.

```
void ISR_EXT0 (void) interrupt 0 using 2
{
    OS_PAUSE_RTOS()          // Disable the RTOS, used in a stand-alone ISR
    /* Our service routine code goes in here */
    /* Our service routine code goes in here */
    /* Our service routine code goes in here */
    OS_RESUME_RTOS()        // Re-enable the RTOS, before exiting the stand-alone ISR
}
```

9.4 PaulOS parameters header file

This is the RTOS parameters header file. We could mainly be needing to set the `TICK_TIMER`, `TICKTIME` and `NOOFTAKS` parameters to reflect out particular application program.

```
/*
*****
* PARAMETERS.H -- RTOS USER DEFINITIONS
*****
*/
#define STACKSIZE      0x0F    // Number of bytes to allocate for the stack
#define CPU            8032    // set to 8051 or 8032
#define TICK_TIMER 2    // Set to 0, 1 or 2 to select which timer to
                        // use as the RTOS tick timer
#define TICKTIME       2      // Length of RTOS basic tick in msec - refer
                        // to the RTOS timing definitions
#define NOOFTASKS     8       // Number of tasks used in application
/*
*****
* PARAMETERS.H -- RTOS USER DEFINITIONS
*****
*/
```

9.5 Example using PaulOS RTOS

This is an example using the PaulOS RTOS. The same function is used to represent 62 different tasks.

Each task would generate random x,y co-ordinates to represent the column (0–79) and row (5–20) where to display a character to represent the task number (A = task 0, B = task 1 and so on). LEDs are connected to Port B (assuming we have the FLT-32 development board) which display the running task number as a binary number. Three other tasks are created to clear the screen, display the stack size used by each task and to generate the random seed. As the program executes, the screen is populated with different characters to represent the 62 tasks.

Excellent Economics and Business programmes at:



university of
 groningen



“The perfect start
 of a successful,
 international career.”

CLICK HERE
 to discover why both socially
 and academically the University
 of Groningen is one of the best
 places for a student to be

www.rug.nl/feb/education

```

/*
*****
*
*           PAULOS
*           The Real-Time Kernel
*
*
*
*           EXAMPLE random06.c
*****
*/
#include <reg52.h>           /* special function registers 8052 */
#include "PaulOS.h"         /* PaulOS C version system calls definitions */
#include <absacc.h>
#include <stdio.h>
#include <stdlib.h>
#include "..\Others\SerP2Pkg.h"
#include "..\Others\Flt32Pkg.h"
#include "..\Others\HYPER_PC.H"

extern uchar MaxSPTs[NOOFTASKS];

#define TaskWaitmSec 900

/*
*****
*
*           TASKS
*****
*/

void CommonTask (void)
{
    uchar x,y,z,s[5];

    z = 1 + OS_RUNNING_TASK_ID();
    OS_PERIODIC_A(0,z,TaskWaitmSec);    /* Run every (1 + Task ID) seconds */
    while(1)
    {
        x = rand()%80;    /* Get X position (0-79) where task number will appear */
        y = 5 + rand()%16; /* Get Y position (5-20) where task number will appear */
        z = OS_RUNNING_TASK_ID();
        PC_DisPChar(y,x,z+'A'); /* Display the task number on the screen */
        WritePort('B',z);    /* LEDs connected to Port B show running task No.*/
        sprintf(s,"%02bu",z);
        PC_DisPStr(22,40,s);
        OS_WAITP();
    }
}
/*
*****
*/

```

```

/*
*****
*/
void StackSize (void) {
    #if STACK_CHECK
        uchar i,j;
        uchar s[20];
        ulong k;

    #endif

    while(1) {
        #if STACK_CHECK

            OS_WAITT_A(0,20,0);

            j = OS_RUNNING_TASK_ID();
            sprintf(s,"%02bu",j);
            PC_DisPStr(22,40,s); /* Display the task number on screen */
            WritePort('B', j);

            for (i=2;i<=20;i++) PC_DisPClr2EndOfRow(i,0);
            PC_DisPStrCntr (2,"Maximum Stack size (per task) used so far");
            PC_DisPStr(3,5,"Task Stack Size Task Stack Size Task Stack Size
Task Stack Size");

            PC_DisPStr(4,5," No Bytes No Bytes No Bytes No Bytes");
            j=0;
            for (i=0;i<=NOOFTASKS;)          {
                sprintf(s," %02bu      %03bu",i,MaxSPTs[i++] - MAINSTACK);
                if(i<=NOOFTASKS+1) PC_DisPStr(5+j,5,s);
                sprintf(s," %02bu      %03bu",i,MaxSPTs[i++] - MAINSTACK);
                if(i<=NOOFTASKS+1) PC_DisPStr(5+j,22,s);
                sprintf(s," %02bu      %03bu",i,MaxSPTs[i++] - MAINSTACK);
                if(i<=NOOFTASKS+1) PC_DisPStr(5+j,41,s);
                sprintf(s," %02bu      %03bu",i,MaxSPTs[i++] - MAINSTACK);
                if(i<=NOOFTASKS+1) PC_DisPStr(5+j++,58,s);
            }

            for (k=0;k<80000;k++){};
            for (i=2;i<=21;i++) PC_DisPClr2EndOfRow(i,0);
            PC_DisPStrCntr (2,"by Paul P. Debono - EXAMPLE Random 06");
            PC_DisPStrCntr (3,"C Version by John Blaut");

        #else

            OS_KILL_IT();

        #endif

    }
}
/*
/*
*****
*/

```

```

void ClearArea (void)
{
    uchar i,s[3];
    OS_PERIODIC_A(0,25,0);          /* Repeat every 25 seconds */
    while(1)
    {
        i = OS_RUNNING_TASK_ID();
        sprintf(s,"%02bu",i);
        PC_DispStr(22,40,s);        /* Display the task number on the screen */
        WritePort('B', i);
        for (i=5;i<=20;i++) PC_DispClr2EndOfRow(i,0);
        PC_DispStr(22,40,s);        /* Display the task number on the screen */
        OS_WAITP();
    }
}

/*
*****
*/
void RandomSeed (void)
{
    uint x;
    uchar z,s[3];
    OS_PERIODIC_A(0,3,500);        /* Run every 3.5 seconds */
    while(1)
    {
        z = OS_RUNNING_TASK_ID();
        sprintf(s,"%02bu",z);
        PC_DispStr(22,40,s);        /* Display the task number on the screen */
        WritePort('B',z);
        x = (x+1)%0xFFFF;
        srand(x);
        OS_WAITP();
    }
}

/*****
/*****
/* $PAGE */
/*****
/***** MAIN
/*****
/* Using ANSI.SYS Escape control sequence */
/* Clear Screen          Esc[2J          */
/* Position Cursor       Esc[row,colH    */
/* Clear to end of line  Esc[K           */
void main (void)
{
    uchar i;

    OS_INIT_RTOS(0x20); /* initialise RTOS variables and stack */
    Init_8255(0x91); /* Initialise the 8255 */
    Set_P2_BaudRate (38400);
}

```

```
PC_DisPClrScr(); /* Clear the screen */
PC_DisPStrCntr (1,"PaulOS, The Real-Time 8051 Co-Operative Kernel");
PC_DisPStrCntr (2,"by Paul P. Debono - EXAMPLE Random 06 with 65 tasks");
PC_DisPStr(22,31,"Task No:");

for(i=0;i<=61;i++)
{
    OS_CREATE_TASK(i,CommonTask); /* CREATE common tasks */
}

OS_CREATE_TASK (62,StackSize); /* CREATE task */
OS_CREATE_TASK (63,ClearArea); /* CREATE task */
OS_CREATE_TASK (64,RandomSeed); /* CREATE task */
OS_RTOS_GO(0); /* Start multitasking */
while (1)
{
    OS_CPU_IDLE();
/* Go to idle mode if doing nothing, to conserve energy */
}

/*
*****
*/
```

LIGS University based in Hawaii, USA

is currently enrolling in the
Interactive Online **BBA, MBA, MSc,**
DBA and PhD programs:

- ▶ enroll **by October 31st, 2014** and
- ▶ **save up to 11%** on the tuition!
- ▶ pay in 10 installments / 2 years
- ▶ Interactive **Online** education
- ▶ visit www.ligsuniversity.com to find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education. More info [here](#).

