10 MagnOS – a Pre-Emptive RTOS

The final RTOS which we discuss is the MagnOS (MAGNus Operating System, Magnus meaning Great in Latin), which gives a demonstration of a pre-emptive RTOS. In this system, each task is given a priority, and the basic control logic of this RTOS is that the highest priority task runs for as long as necessary, until a higher priority task becomes ready to execute. The trick here is to learn to decide what priority to give to the individual tasks so as to avoid having a single task take over completely the processor time, without giving a chance for other tasks to run.

This RTOS, which is a pre-emptive RTOS is a further modification of the PaulOS co-operative RTOS program, written in C to make it more versatile and easier to port to other micro-controller variations and types. It can be further developed into a more complex RTOS if one can dedicate more time to it. The original idea behind this RTOS once again came from the book by Prof. Thomas W. Schultz "C and the 8051 – Volume II"[2] which described the basic ideas and workings of such a simple but effective pre-emptive RTOS.

Obviously, the main improvement of this RTOS over the PaulOS RTOS, is in the pre-emptive swapping of tasks capability. If the RTOS sees any task which is ready to execute and which has a higher priority than the current one running, it will interrupt that running task and it will start (or resume) executing the higher priority task instead. The task will then continue to run until it either gives up the processor/controller time on its own accord by some command similar to the PaulOS method (such as OS_WAITT(x)) or else another task having an even higher priority becomes ready to execute and therefore the RTOS would give it the priority to run.

Great care has to be taken in deciding what priority to allocate to each of the individual tasks and also in the use of variables and/or resources by more than one task.

Naturally there are some memory space and speed penalties to pay for this versatility. Because of this, if one can perform the required project with a co-operative RTOS, then one has no need for the pre-emptive RTOS. However the improvements more than outweigh the penalties, and since it is written in C, the student can better understand the workings of the RTOS. The full source code listing of this RTOS can be found in appendix E. Here is the list of the MagnOS RTOS commands and some description of each:

10.1 MagnOS System Commands

Some of the commands are exactly the same as those used in the PaulOS RTOS, but are also being listed here for the sake of completeness.

The following RTOS system calls do not receive any parameters:

- OS_RTOS_GO (void); // Starts the RTOS with priorities if required
- OS_WAITP (void); // Waits for end of task's periodic interval
- OS_RUNNING_TASK_ID(void); // Returns the number (unsigned char) of current task

The following RTOS system calls do receive parameters :

- OS_INIT_RTOS (uchar iemask); // Initialises all RTOS variables
- 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_PERIODIC(uint ticks); // Set task to repeat periodically

- OS_CHECK_TASK_PRIORITY (uchar task_num) // gets the requested task priority setting
- OS_CHANGE_TASK_PRIORITY (uchar task_num, uchar new_prio) // sets the task priority

- OS_RELEASE_RES (uchar Res_Num) // releases the resource, for use by other tasks
- OS_WAIT4RES (uchar Res_Num, uint ticks) // wait for the resource within ticks time

- OS_SEND_MSG (struct letter xdata *msg) // send a message to a task
- OS_CLEAR_MSG (struct letter xdata *msg) // clears the message
- OS_CHECK_MSG (struct letter xdata *msg) // checks if message is present
- OS_GET_MSG (struct letter xdata *msg) // gets the message
- OS_WAIT_MESSAGE (struct letter xdata *msg) // waits for a message

- OS_CHECK_TASK_SEMA4 (uchar task_num) // checks the semaphore
- OS_SEMA4MINUS (uchar task_num, uchar units) // deducts units from the semaphore
- OS_SEMA4_PLUS (uchar task_num, ucahr units) // adds units to the semaphore
- OS_WAIT4SEM (uint ticks) // waits for the semaphore to get to zero within ticks time

- OS_KILL_TASK (uchar tasknum); // Kills a task – sets it to wait forever
- OS_CREATE_TASK (uchar tasknum, uint taskadd, uchar priority); // Creates a task

10.2 Detailed description of commands

This pre-emptive RTOS (source listing given in the appendix E) provides some variations and additional commands which were implemented after having used the first test versions of the program for some time. We now describe what these commands actually do and how they were implemented. Although we might be repeating ourselves the commands which are very similar to PaulOS are once again described here since they might have some slight changes due to the priority and pre-emptive components of the MagnOS RTOS. Moreover it eliminates the need to continuously flick over the pages for references. Later, after reading this chapter, one can refer back to the PaulOS Chapter 9, and to the source code and the remarks in appendices D and E for further explanations and comparisons.

Each task has its own set of parameters, as declared in MagnOS.h file (see Appendix E) and shown here, since some of these parameters are used when explaining the commands in the various 10.2.x sub-sections.

```
struct task_param {                 /* 13 bytes + 13 registers + stack per task */
        uchar catalog;              /* task id */
        uchar status1;              /* status flags, see below for details */
        uchar status2;              /* status flags, see below for details */
        uchar priority;             /* priority number, low = high priority */
        uchar semaphore;            /* counting semaphore for each task */
        uchar resource;             /* resource number required */
        uchar stackptr;             /* stack pointer SP storage location */
        uchar intnum;               /* task waiting for this interrupt number */
        uint timeout;               /* task waiting for this timeout in ticks, */
                                    /* 0 = not waiting */
        uint interval_count;        /* time left to wait for this periodic */
                                    /* interval task in ticks */
        uint interval_reload;       /* periodic tick interval reload value */
        uchar rega;                 /* registers storage area, ready for context */
                                    /* switching use */
        uchar regb;
        uchar rdph;
        uchar rdpl;
        uchar rpsw;
        uchar reg0;
        uchar reg1;
        uchar reg2;
        uchar reg3;
        uchar reg4;
        uchar reg5;
        uchar reg6;
        uchar reg7;
        char stack[STACKSIZE];          /* stack storage area */
                };
```

10.2.1 OS_RTOS_GO (void)

This is the command which starts the RTOS going.

It performs the following:

- It loads the correct timer (selected by the parameter TICK_TIMER) with correct reload value so as to generate overflow interrupts according to the BASIC_TICK selected.
- Starts the timer and sets the timer overflow interrupt flag so that it would cause an interrupt immediately.
- Signals to the RTOS that there are tasks in the READY queue, by setting flag TinQFlag to 1.
- Enables global interrupts, so that the timer interrupt is recognised immediately.

Such a command is used only once, normally in the main() function so as to start the RTOS going. An example of its usage is shown below:

```
OS_RTOS_GO();
```

10.2.2 OS_PERIODIC (uint ticks)

This command initalises 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 10.2.3. An example of its usage is also given in that same sub-section.

10.2.3 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 with the highest priority 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
}
```

10.2.4 OS_RUNNING_TASK_ID(void)

This command simply returns the number (id) of the task which is currently running. This command does not cause a voluntary task change. Its usage is very straight forward, assuming that CurrentTask was previously declared as an *unsigned char* variable:

```
CurrentTask = OS_RUNNING_TASK_ID();
```

10.2.5 OS_INIT_RTOS (uchar iemask)

This command initalises all the RTOS and tasks variables, stacks, interrupt masks to their default values (mostly zeroes). The iemask is used to enable the interrupts which one intends to use for the tasks and the RTOS tick timer itself. The iemask bits reflect the interrupt numbers used in the 8051as shown in Table 10-1.

Bit	7	6	5	4	3	2	1	0
Interrupt	NA	NA	Timer 2	Serial	Timer 1	Ext 1	Timer 0	Ext 0

Table 10-1 IEMASK values

It is normally the first RTOS command used at the very beginning of the main() function, before creating the tasks and its use is shown below:

```
OS_INIT_RTOS(0x25); // mask=00100101, use interrupts EXT0, EXT1 and TF2
                    // Timer 2 would presumably be the RTOS tick timer
```

10.2.6 OS_WAITI (uchar interrupt)

This command causes the task to wait for the required interrupt. The task goes into a waiting state and the next ready task with the highest priority takes over. The interrupt cannot be the same as that being used for the tick timer and obviously there cannot be a stand-alone ISR routine which is being activated this interrupt.

It performs the following:

- Sets the corresponding 'waiting for interrupt' flag for the task.
- Stores environment in preparation for the voluntary task swap.
- Performs the task swap.

It can be used by a task as follows:

```
OS_WAITI(4); // wait for the serial interrupt number 4
```

10.2.7 OS_WAITT (uint ticks)

This command causes the task to wait for the required number of ticks. The task goes into a waiting state and the next ready task with the highest priority takes over.

It performs the following:

- Sets the corresponding 'timeout waiting parameter' for the task.
- Stores environment in preparation for the voluntary task swap.
- Performs the task swap

It can be used by a task as follows:

```
OS_WAITT(100); // wait for 100 ticks to pass
```

10.2.8 OS_CHECK_TASK_PRIORITY (uchar task_num)

This command returns the value (type unsigned char) of the required task priority. This command does not cause a voluntary task change and is used as follows:

```
Task_5_Priority = OS_CHECK_TASK_PRIORITY(5);
```

where Task_5_Priority would be a previously declared variable of type *unsigned char*.

It is mainly used in a task so as to be able to store the task priority before changing it. A higher value indicates a higher priority, which is the opposite of what was the practice in PaulOS.

10.2.9 OS_CHANGE_TASK_PRIORITY (uchar task_num, uchar new_prio)

This command changes the value of the required task priority. This command does not cause a voluntary task change and is normally used so as to increase or reset the priority of a particular task. This is normally required when for example something which cannot be interrupted is about to be executed, in which case the priority is temporary set to the highest value until the time critical code is executed. The task priority can then be restored to the original value. Such a task would be coded as follows:

```
P = OS_CHECK_TASK_PRIORITY(thistask);
OS_CHANGE_TASK_PRIORITY(thistask, highest_priority);
……..    // time critical code here
……..
OS_CHANGE_TASK_PRIORITY(thistask, P);
……..
```

10.2.10 OS_RELEASE_RES (uchar Res_Num)

This command frees the given resource, thus making it available to other tasks. This command may cause a voluntary task change if there was another higher priority task waiting for this resource. The resource can be used to represent a function, a device or a variable.

The command performs the following:

- Stores environment in preparation for the voluntary task swap.
- Checks all tasks in order to see which is the highest priority task that was waiting for this resource.
- If no other higher priority task was waiting for this resource, then it simply marks this resource as being free and exits without performing any task swap.
- If a higher priority task is found, then:
  - the resource is marked as being used by the new task.
  - The current task is placed in the waiting queue (code default is for 3 ticks, the number was chosen for no particular reason).
  - Performs the task swap.

It can be used by a task as follows:

```
OS_RELEASE_RES(10);        // release resource number 10, which could represent a printer
                           // for example.
```

10.2.11 OS_WAIT4RES (uchar Res_Num, uint ticks)

This command causes the task to wait for the required resource to become available within a given timeout. A ticks value of zero implies keeping on waiting for the resource forever. The task goes into a waiting state if the resource is not available and the next ready task with the highest priority takes over. If the resource is already free and available, the task simply continues to execute. This command therefore may or may not perform a task swap.

This command performs the following:

- Stores environment in preparation for the voluntary task swap, if needed.
- If the resource is already available, it simply marks the resource as being in use by this task and exits without performing any task swap.
- If the resource is being used by some other task:
  - It sets the flags indicating that the task is waiting for a resource.
  - Marks which resource it is waiting for.
  - Sets the timeout tick time.
  - Performs the voluntary task change.

The usage of this command is as follows:

```
        OS_WAIT4RES(10,0);   // wait forever for resource number 10
or
        OS_WAIT4RES(10,80); // wait for a maximum of 80 ticks for resource number 10,
                            // it will be placed in the ready queue once resource is available,
                            // if resource is still not available after 80 ticks, the task will
                            // still go ahead, which might be catastrophic!
```

10.2.12 OS_SEND_MSG (struct letter xdata *msg)

This command sends a message to another task. There are two message arrays stored in external RAM area associated with messages:

- A message (msg) array where messages are written to when a task wants to send a message. This same array is also used when a task reads a message.
- A mailbox (mbox) array where messages are stored whilst waiting to be read by the destination task.

If the other task was already waiting for this message, then a voluntary task change is invoked. The message format (structure) carries information regarding the sender, recipient and data of the message itself. The structure of the message is composed of bytes representing the destination, source, length of message and up to 16 bytes for the message itself as shown in the declarations below:.

```
union dataformat {struct{ulong HI1,LO1,HI0,LO0;}dblwords;
                  struct{uint Hi3,Lo3,Hi2,Lo2,Hi1,Lo1,Hi0,Lo0;}words;
                  struct{uchar hi7,lo7,hi6,lo6,hi5,lo5,hi4,lo4,hi3,lo3,hi2,lo2,hi1,lo1,hi0,lo0;}bytes;
                  struct{char s[DATASIZE];}string;};

struct letter{uchar dest,src,len; union dataformat dat;};
```

The 'union' is used so that the data can be accessed easily in any form.

The command does the following:

- Stores the environment in preparation for the voluntary task swap, if needed.
- Goes through the mbox array and if it finds another task already waiting for this message:
  - Copies the message for that task on to the msg array.
  - Moves the task waiting for message into the ready queue.
  - Places the task sending the message into the waiting (for 1 tick) queue. It would then be placed in the normal ready queue automatically by the RTOS at the next tick.
  - Performs the voluntary task change.

- If there is no other task currently waiting for this message, it simply leaves the message in the mbox array and exits without performing any task swap.

The usage of this command is as follows:

```
OS_SEND_MSG(letter1);
```

10.2.13 OS_CLEAR_MSG (uchar task_num)

This command simply clears a message from the mbox array, destined for the particular task number. This command does not cause a voluntary task change.

The usage of this command is as follows:

```
OS_CLEAR_MSG(3);    // clear message destined for task number 3
```

Please not that this does not cater for multiple messages to the same task.

10.2.14 OS_CHECK_MSG (uchar task_num)

This command checks if there is a message destined for a particular task. It returns a bit value of 1 if the message is present and a bit value of zero if there is no message. This command does not cause a voluntary task change.

The use of this command is as follows:

```
Bit_Message_Present = OS_CHECK_MSG(3);    // checks if there is message for task 3
```

where Bit_Message_Present would be declared of type *bit*.

10.2.15 OS_GET_MSG (struct letter xdata *msg)

This command reads the message destined for the current task. This command does not cause a voluntary task change.

This command does the following:

- It goes through the mailbox mbox array and copies the message destined to it on to the message msg array of the current task.
- It then clears the mailbox.

The use of this command is as follows:

```
OS_GET_MSG(letter3);          // get the message for task number listed within the
                              // letter3 variable (type struct letter)
```

10.2.16 OS_WAIT_MESSAGE (struct letter xdata *msg, uint ticks)

This command waits for a message within ticks time. If the ticks parameter is set to zero, then the task would wait until the message is received, whenever that may occur. If the message is not already present, then a voluntary task change is performed and the next ready task with the highest priority takes over. If on the other hand the message is already present, no task swap is performed and the message is transferred from the mbox to the msg array of the task.

This command performs the following:

- Saves task environment in preparation for the expected voluntary task swap.
- It checks the mbox array and if the message is already present:
  - it reads the message to the msg array.
  - clears the mbox and exits.

- If the message is not present, then it has to wait for it and therefore
  - It searches the mbox for a free location and reserves that area in mbox for the message to be received and performs the task swap, waiting for the message within the specified ticks.
  - If no free location is found in mbox it exits, marking the NO_MBOX_FREE_F flag and exits without performing the task swap.

This command can be used as follows:

```
OS_WAIT_MESSAGE(letter,0);
if(task[Running].status2 & NO_MBOX_FREE_F)== NO_MBOX_FREE_F) {…..}
        // implies no free storage space found in mailbox
```

For a full explanation of status2, please refer to Appendix E header file MagnOS.h.

10.2.17 OS_CHECK_TASK_SEMA4 (uchar task_num)

This command checks for a semaphore of the particular task and simply returns an *unsigned char* value of the semaphores left. Each task can have its own semaphore, stored in its parameters array (sec section 10.2). This command does not cause a voluntary task change.

This command can be used as follows:

```
SemaphoresLeft = OS_CHECK_TASK_SEMA4(5); // checks the semaphores left for task 5
```

where SemaphoresLeft is of type *unsigned char*.

10.2.18 OS_SEMA4MINUS (uchar task_num, uchar units)

This command deducts the given number of units (normally 1) from a semaphore of the particular task number. This command causes a voluntary task change only if the semaphores for the required task drop down to zero after the subtraction takes place. It therefore performs the following:

- Saves task environment in preparation for the eventual voluntary task swap.
- Deducts the required number of semaphore units (final resultant semaphores will be zero or greater).
- If the semaphore is now zero and there was a task waiting for the semaphore (to reduce to zero) then:
  - The task which was waiting for the semaphore is now placed in the waiting queue (the usual 1 tick time waiting delay, then placed automatically in the ready queue by the RTOS at the next tick) after clearing its semaphore waiting flag.
  - The present task is placed in the waiting queue (for 5 ticks, so as to give some time to other tasks. This can be changed in the source code).
  - A task swap is performed.

This command is used as follows:

```
OS_SEMA4MINUS(4,1);          // deduct 1 unit from the semaphore of task number 4
```

10.2.19 OS_SEMA4_PLUS (uchar task_num, ucahr units)

This command simply adds the given number of units (normally 1) to a semaphore of the particular task number. This command does not cause a voluntary task change.

This command is used as follows:

```
OS_SEMA4PLUS(4,1);           // add 1 unit to the semaphore of task number 4
```

10.2.20 OS_WAIT4SEM (uint ticks)

This command causes the task to wait for its semaphore to reach a value of zero within a given timeout period. A timeout ticks of zero implies having to wait forever until the semaphore reaches zero. The task goes into a waiting state if the semaphore is not zero and the next ready task with the highest priority takes over.

The command does the following:

- Saves task environment in preparation for the eventual voluntary task swap.
- If the semaphore is already zero, it clears the wait for semaphore flag and resumes execution.
- If however the semaphore is not yet zero it sets the wait for semaphore flag and the wait for timeout flag and then it performs a voluntary task change.

The command can be used as follows:

```
OS_WAIT4SEM(0);                // wait forever until the semaphore becomes zero
```

10.2.21 OS_KILL_TASK (uchar tasknum)

This command kills the specified task and it will not execute again. This command will cause a voluntary task change. It performs the following:

- If the task was already killed by some other task, it simply exits.
- Otherwise it
  - Marks it as killed by setting the TASK_KILLED_F flag.
  - Clears and frees any mbox messages intended for this task.
  - If the task happens to be the one currently running (the task wants to commit suicide!), it clears all its flags and sets the timeout to zero so that it will appear to be waiting for ever. A task change is not called, without the need to save the environment.
  - If the task happens to be in the Ready queue, then the task number is changed to that of the idle task. Any multiple idle tasks entries in the queue are eliminated, so that at the end we would have only one idle task in the Ready queue.
  - All the task flags are reset.

This command can be used as follows:

```
OS_KILL_TASK(4);               // kill task number 4
```

10.2.22 OS_CREATE_TASK (uchar task_num, uint task_add, uchar task_priority)

This last command in the list creates a task and is used in the main task after intialising the RTOS but before starting the RTOS. This command does not cause a voluntary task change. The command does the following:

- It increments the Ready queue pointer and stores the task number in the Ready queue.
- It places the task start address in the stack area of that task, which would ultimately end in the SP once a task change is performed.

- Stores the priority and other flags in the status area of that task and exits. A zero value would represent a low priority, while a value of 255 would represent the highest (top) priority.
  The command is normally used in the main() function, once for every task that needs to be created:

```
OS_CREATE_TASK(2,task_two,5);
```

// creates task of function named task_two, giving it a task number of 2 with a priority of 5.

MagnOS Parameters.h header file

This is the header file which we would need to modify depending on the application program. Typically we would need to set the TICKTIME and NOOFTASKS variables so as to reflect the actual tick time (say 1, 10 or 50 milliseconds) and number of tasks which we have in our main program.

```
/*
************************************************************************
*                 PARAMETERS.H --- RTOS KERNEL HEADER FILE
*
* For use with MagnOS_V01.C
* Co-Operative RTOS written in C by Ing. Paul P. Debono
*                 for use with the 8051 family of microcontrollers
*
* File      : Parameters_V01.H
* Revision  : 8
* Date      : February 2006
* By        : Paul P. Debono
*
*                 B. Eng. (Hons.) Elec. Course
*                 University Of Malta
*
************************************************************************
*/

/*
************************************************************************
*                                       RTOS USER DEFINITIONS
************************************************************************
*/

#define STACKSIZE   0x10
            // Max size of stack for each task - no need to change
#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    50   // Length of RTOS basic tick in msec
                         // - refer to the RTOS
                         // timing definitions
#define NOOFTASKS   6    // Number of tasks used in the application program

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

We now give an example using the MagnOS RTOS just for demonstration purposes.

- Task 0 runs every minute and displays the alphabet in upper case letters, priority 1.
- Task 1 runs every four and a half secomds and displays the alphabet in lower case letters, priority 2.
- Task 2 runs every 700 milliseconds and displays the alphabet in Capital letters, priority 5.

It is interesting to change the periodicity (in the task functions) and priorities (in the main program) of the tasks and see the effect on the overall performance of the program. We need to remember that as far as the priority is concerned, a 0 value represents the lowest priority, and 255 would represent the highest (top) priority available.

Certain values can cause the RTOS to fail to start/finish the tasks within the required intervals. Scheduling probl;ems arise and the reader is urged to read material on task scheduling and assigning priorities for pre-emptive RTOSs.

```
/*********************************************************************/
/*                                                                   */
/*    MagnosTest_00.c: Demo using pre-emptive Tasks                  */
/*                                                                   */
/*                                                                   */
/*********************************************************************/

#include <reg52.h>            /* special function registers 8052      */
#include <MagnOS_V01.h>       /* RTOS system calls definitions        */
#include <absacc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "UART_REENTRANT.h"

/*********************************************************************/
/*    Task 0:                                                        */
/*********************************************************************/

void CAPS (void){             /* Prints CAPITAL Letters               */
char code msg[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ-ZYXWVUTSRQPONMLKJIHGFEDCBA";

            static unsigned long i;
            static unsigned int j;

            OS_PERIODIC_A(1,0,0);    /* runs every 1 minute          */
            while(1)
            {
            TX_STRING("\nRunning Task: ");
            TX_CHAR('0'+ OS_RUNNING_TASK_ID());
            TX_STRING(" Priority: ");
            TX_CHAR ('0' + OS_CHECK_TASK_PRIORITY(OS_RUNNING_TASK_
ID()));
            TX_STRING("\n\r");
            j=0;
                  while (msg[j]!='\0')
                    {
                        TX_CHAR(msg[j++]);
                        for (i=1;i<10000;i++);
                    /* just a delay to simulate long process */
                    }
            TX_STRING("\n\r");
            OS_WAITP();
            }
 }

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

```
/*********************************************************************/
/* Task 1: */
/*********************************************************************/

void Small (void){ /* Prints small Letters */
char code msg[]="abcdefghijklmnopqrstuvwxyz-zyxwvutsrqponmlkjihgfedc-
ba";

             static unsigned long i;
             static unsigned int j;

             OS_PERIODIC_A(0,4,500);       /* runs every 4.5 seconds */
             while(1)
            {
             TX_STRING("\nRunning Task: ");
             TX_CHAR('0'+ OS_RUNNING_TASK_ID());
             TX_STRING(" Priority: ");
             TX_CHAR ('0' + OS_CHECK_TASK_PRIORITY(OS_RUNNING_TASK_
ID()));
             TX_STRING("\n\r");
             j=0;
                   while (msg[j]!='\0')
                    {
                     TX_CHAR(msg[j++]);
                     for (i=1;i<300;i++);
                     /* just a delay to simulate long process */
                    }
             TX_STRING("\n\r");
             OS_WAITP();
      /* wait for Periodic timeout */
             }
 }

/*********************************************************************/
/*    Task 2:                                                        */
/*********************************************************************/

void Numbers (void){ /* Prints Numbers */
char code msg[]="0 1 2 3 4 5 6 7 8 9 - 9 8 7 6 5 4 3 2 1 0";

            static unsigned long i;
            static unsigned int j;
```

```
            OS_PERIODIC_A(0,0,700);  /* runs every 700 milliseconds */
            while(1)
            {
            TX_STRING("\nRunning Task: ");
            TX_CHAR('0'+ OS_RUNNING_TASK_ID());
            TX_STRING(" Priority: ");
            TX_CHAR ('0' + OS_CHECK_TASK_PRIORITY(OS_RUNNING_TASK_
ID()));
            TX_STRING("\n\r");
            j=0;
                   while (msg[j]!='\0')
                    {
                     TX_CHAR(msg[j++]);
                     for (i=1;i<100;i++);
                    /* just a delay to simulate long process */
                    }
                   TX_STRING("\n\r");
                   OS_WAITP();      /* wait for Periodic timeout */
            }
 }

/*********************************************************************/
/*    Main: Initialise and CREATE tasks */
/*********************************************************************/

void main (void)        {     /* program execution starts here        */
      INIT_SERIAL_T1(57600);

      TX_STRING("Initialising MagnOS Pre-Emptive RTOS\n\r");
       OS_INIT_RTOS(0x20);/* initialise MagnOS RTOS variables and stack */
                                 /* using Timer 2 interrupts       */
/* A HIGH PRIORITY NUMBER, MEANS A HIGH PRIORITY TASK */
      OS_CREATE_TASK(0, CAPS,       1);   // priority 1
      OS_CREATE_TASK(1, Small,      2);   // priority 2
      OS_CREATE_TASK(2, Numbers,    5);   // priority 5

      TX_STRING("Tasks Created, running MagnOS RTOS\n\r");
      OS_RTOS_GO();                       /* start RTOS */

      while (1)
      {
       OS_CPU_IDLE();
      }
}
/*********************************************************************/
```