Semiconductors

AN10413

µC/OS-II Time Management in LPC2000

Rev. 01 — 15 December 2005 **Application note**

Document information

Info	Content
Keywords	µC/OS-II, MCU, ARM, LPC2000, Timer, IRQ, VIC
Abstract	This application note demonstrates how to implement µC /OS-II time management in LPC2000 microcontroller family from Philips Semiconductors. In addition to perform time management of µC/OS-II, a simple demo code is given. All together, the note offers users a quick start in using µC/OS-II time management in LPC2000.

PHILIPS

Revision history

Rev	Date	Description
01	20051215	Initial version

Contact information

For additional information, please visit: http://www.semiconductors.philips.com

For sales office addresses, please send an email to: sales.addresses@www.semiconductors.philips.com

1. Introduction

µC/OS-II (Pronounced “Micro C O S 2”), which stands for Micro-Controller Operating System Version 2, is a type of real-time operating system. Because of its real-time kernel, ease to port, use and reliability, it is widely used in all kinds of applications such as cameras, medical instruments, engine controls, ATM machines and many more. µC/OS-II can run on most 8/16/32-bit microprocessors or microcontrollers.

Time management is one important part of µC/OS-II. It provides periodic interrupts to keep track of time delay and timeouts. The periodic time is called Clock Tick. This interrupt can be viewed as the system's heartbeat. Usually, a tick should occur between 10 and 100 times per second, or Hertz. The faster the tick rate, the higher the overhead imposed on the system. The actual frequency of the clock tick depends on the desired tick resolution of user application. Usually the tick source is obtained from a hardware timer.

The LPC2000 family is based on the 16/32-bit ARM7TDMI-S™ microcontroller. All the part numbers can support µC/OS-II. In LPC2000 family, two 32-bit timers/counters are provided. Both can be used as the source of the tick. Here we use timer0 as an example and timer0 will be configured to trigger an IRQ interrupt periodically. The code is developed in ADS (ARM Development Suite) v1.2 and most written in ANSI C. The code was tested on an evaluation board with LPC2129, which uses a 12 MHz crystal.

2. Initialization

2.1 Exception vector table

LPC2000 family is based on a 16/32-bit ARM7TDMI-S™ CPU. The ARM CPU contains an exception vector table, which is used to support seven types of exception. When an exception occurs, an execution is forced from a fixed memory address corresponding to the type of exception. The exception vector table for the ARM is shown in Table 1:

Table 1: Exception vector table

Exception	Mode	Vector Address
Reset	SVC	0x00000000
Undefined Instruction	UND	0x00000004
Software Interrupt (SWI)	SVC	0x00000008
Prefetch abort	Abort	0x0000000C
Data abort	Abort	0x00000010
-	-	0x00000014
IRQ (Normal Interrupt)	IRQ	0x00000018
FIQ (Fast Interrupt)	FIQ	0x0000001C

When on reset, the CPU begins executing from the reset vector entry, then jumps to initialization subroutine, starting system setting. The startup code is written in assembly code as shown below:

Startup code

```
;Imported external symbols declaration
        IMPORT  Reset
        IMPORT  FIQHandler_C
; /*************************************
; Exception Vectors
; *************************************/
     CODE32
AREA  StartUp, CODE, READONLY
     ENTRY
Vectors
        LDR     PC, ResetAddr
        LDR     PC, UndefinedAddr
        LDR     PC, SWI_Addr
        LDR     PC, PrefetchAddr
        LDR     PC, DataAbortAddr
        DCD     0xb9205f80
        LDR     PC, [PC, #-0xff0]            ;for vectored and non-vectored IRQ
        LDR     PC, FIQ_Addr

ResetAddr           DCD     Reset
UndefinedAddr       DCD     Undefined
SWI_Addr            DCD     Swi
PrefetchAddr        DCD     PrefetchAbort
DataAbortAddr       DCD     DataAbort

FIQ_Addr            DCD     FIQ_Handler

;/*************************************
;Undefined instruction exception handler
;*************************************/
Undefined
        b       Undefined
;/*************************************
;Swi exception handler
;*************************************/
Swi
        b       Swi
;/*************************************
;Prefetch abort exception handler
;*************************************/
PrefetchAbort
        b       PrefetchAbort
;/*************************************
;Data abort exception handler
;*************************************/
DataAbort
        b       DataAbort
;/*************************************
;FIQ exception handler
;*************************************/
```

```
FIQ_Handler
        STMFD   SP!, {R0-R3, LR}
        BL      FIQHandler_C            ;call the FIQ ISR subroutine
        LDMFD   SP!, {R0-R3, LR}
        SUBS    PC, LR,  #4
    END
```

Actually, the given handlers in the startup code do not do anything useful. They are setup here just for completeness. You can implement them according to your application.

2.2 System config

System config such as PLL, VPBDIV and MAM is performed in C code. The code is tested on an evaluation board, which uses a 12 MHz crystal. To make CPU run at full speed of 60 MHz, PLL is set to 5. And VPB is set to a quarter of CPU speed. Using Memory Map Register, you can remap interrupt vectors to 0x0000 0000-0x000 0001c (on-chip flash), 0x4000 0000-0x4000 001c (on-chip ram) or 0x8000 0000-0x8000 001c (external memory, only for LPC22xx). The system initialization code is shown below:

```
#define  PLL_PLLE           1          //PLL enable (1)or disable(0)
#define  PLL_PLLC           1          //PLL connect(1) or disconnect(0)
#define  PLL_M              5          //PLL Multiplier  value
#define  PLL_P              1          //PLL divider value: p
#define  VPB_DIVIDER        0          //the divider of VPB

/* System Initialization           */
void InitLPC2000(void) {
      WDMOD=0;                    //disable WDT

      VICIntEnClr=0xffffffff;     //disable all interrupts
      VICVectAddr=0;
      VICIntSelect=0;

      /* PLL configuration  */
      if(PLL_PLLE){
         PLLCFG=(PLL_M- 1) | (PLL_P << 5);
         PLLCON=PLL_PLLE;
         PLLFEED = 0xAA;
         PLLFEED = 0x55;
         while((PLLSTAT & (1 << 10)) == 0);      // Wait for PLL lock

         PLLCON=PLL_PLLE|PLL_PLLC<<1;            //connect PLL
         PLLFEED = 0xaa;
         PLLFEED = 0x55;
      }

      VPBDIV=VPB_DIVIDER;              //peripheral clock config

      /* MemRemap Config  */
#ifdef __Ram_Mode
      MEMMAP = 0x2;          //remap to 0x40000000
#endif
```

```
#ifdef __Flash_Mode
      MEMMAP = 0x1;          //remap tp 0x0
#endif

#ifdef __ExtMem_mode
      MEMMAP = 0x3;          //remap to 0x80000000, only for lpc22xx
#endif
}
```

2.3 Timer Initialization

Timer0 is configured to generate clock tick. The tick frequency is defined as **OS_TICKS_PER_SEC** in file os_cfg.h. Timer0 counter is set according to tick frequency and peripheral clock.

LPC2000 family contains VIC (Vectored Interrupt Controller) that supplies a vector (i.e., an address) for each interrupt source. VIC can take up to 32 interrupt request inputs and programmably assign them into three categories, FIQ, vectored IRQ and non-vectored IRQ. FIQ requests have the highest priority. Vectored IRQs have the intermediate priority, but only 16 of the 32 requests can be assigned to this category. Non-vectored IRQs have the lowest priority.

Each peripheral device has one interrupt line connected to the VIC, but may have several internal interrupt flags. Fig 1 lists the interrupt sources for each peripheral function.

VICIntEnable register controls which of the 32 interrupt requests contributes to FIQ or IRQ and enables it. **VICVectCntx** and **VICVectAddrx** registers control one of 16 vectored IRQ slots together. **VICVectCntx** Register selects the interrupt source and **VICVectAddrx** register holds the address of ISR of the corresponding vectored IRQ.

As shown in the exception vector table (Table 1:), when an IRQ occurs, ARM CPU will redirect code execution to the address specified at location 0x0000 0018. For vectored and non-vectored IRQ's the following instruction could be placed at 0x18:

LDR pc, [pc,#-0xFF0]

This instruction loads PC with the address that is present in **VICVectAddr** register then gets the IRQ service routine from **VICVectAddr** register and jumps to the value read.

Block	Flag(s)	VIC Channel #
WDT	Watchdog Interrupt (WDINT)	0
-	Reserved for software interrupts only	1
ARM Core	Embedded ICE, DbgCommRx	2
ARM Core	Embedded ICE, DbgCommTx	3
TIMER0	Match 0 - 3 (MR0, MR1, MR2, MR3) Capture 0 - 3 (CR0, CR1, CR2, CR3)	4
TIMER1	Match 0 - 3 (MR0, MR1, MR2, MR3) Capture 0 - 3 (CR0, CR1, CR2, CR3)	5
UART0	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI)	6
UART1	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI) Modem Status Interrupt (MSI)	7
PWM0	Match 0 - 6 (MR0, MR1, MR2, MR3, MR4, MR5, MR6)	8
I^2C	SI (state change)	9
SPI0	SPI Interrupt Flag (SPIF) Mode Fault (MODF)	10
SPI1	SPI Interrupt Flag (SPIF) Mode Fault (MODF)	11
PLL	PLL Lock (PLOCK)	12
RTC	Counter Increment (RTCCIF) Alarm (RTCALF)	13
System Control	External Interrupt 0 (EINT0)	14
System Control	External Interrupt 1 (EINT1)	15
System Control	External Interrupt 2 (EINT2)	16
System Control	External Interrupt 2 (EINT2)	17
A/D	A/D Converter	18
CAN	CAN and Acceptance Filter 1 ORed CAN, LUTerr int CAN1 and CAN2: 2x(Tx int, Rx int) LPC2119/2129/2292/2294 CAN3 and CAN4: 2x(Tx int, Rx int) LPC2194/2292/2294 only	19 20-23 24-27

Fig 1. Connection of interrupt sources to VIC

Here Timer0 interrupt is configured as a vectored IRQ interrupt and the priority is set to 15. The initialization code can be as following:

```
#define OS_TICKS_PER_SEC          50          //Set the number of ticks in one second

void TIMER0_InitTimer(void) {
      TIMER0_IR  = 0xff;                      //clear interrupts
      TIMER0_TC  = 0;
      TIMER0_MCR = 0x03;                      //reset and interrupt on match
      TIMER0_MR0 = (FPCLK/ OS_TICKS_PER_SEC);         //set the match value
```

```
    //Initialize timer0 interrupt
    VICIntEnClr =  (1 << 4);                        //disable timer0 interrupt
    //config timer0 interrupt as the lowest v-IRQ
    VICVectAddr15 = (LPC_INT32U)IRQASMTimer0;       //set timer0 ISR address
    VICVectCntl15 = (0x20 | 0x04);
    VICIntEnable = (1 << 4);                        //enable timer0 interrupt

    TIMER0_TCR = 0x01;                              //enable timer0 counter
}
```

In µC/OS-II, one thing you have to notice is that you must enable tick interrupts after multitasking has started, i.e. after calling OSStart(). In other words, you should initialize and enable tick interrupts in the first task that executes following a call to OSStart(). A common mistake is to enable ticker interrupts between calling OSInit() and OSStart() as shown in the following code. Because at that point, µC/OS-II is in an unknown state and it will cause your application to crash.

```
void main(void) {
   …
   OSInit();                                    // initialize uC/OS II
   …
   /* user application initialization code */
   /* create application task by calling OSTaskCreate() */
   …
   Enable Ticker Interrupts;                    //DO NOT DO THIS HERE!!!
   …
   OSStart();                                   // start multitasking
}
```

3. Clock Tick ISR

3.1 ISR in µC/OS-II

In µC/OS-II, ISRs includes several parts: save CPU registers, call function OSIntEnter(), execute user code, call function OSIntExit(), restore CPU registers and return.

Function OSIntEnter() is used to notify µC/OS-II that you are about to service an interrupt (ISR) and function OSIntExit() is used to notify µC/OS-II that you have completed serving an ISR. With OSIntEnter() and OSIntExit(), µC/OS-II can keep track of interrupt nesting and thus only perform rescheduling at the last nested ISR.

Sometime when the last nested ISR is completed, the interrupted task will be no longer the task that needs to run because a new, higher-priority task is now ready. In this case, interrupt level context switch is needed. This is done by function _IntCtxSw(). Then after return, the new, higher-priority task is running and the old one is pending.

These codes should be written in assembly language because you cannot access CPU registers directly from C. But user code can be written in C. Here we use macro code to implement ISR in file irq_handler.s. The code can be shown as following and should be duplicated for each ISR you have in your system.

```
        MACRO
```

```
$IRQ_AsmEntery HANDLER $IRQ_CEntry

$IRQ_AsmEntery
        stmfd sp!,{r0-r3,r12,lr}            ; push r0-r12 register file and lr

        bl OSIntEnter                       ; Interrupt Nest++
        bl $IRQ_CEntry                      ; User ISR Subroutine
        bl OSIntExit

        ldr r0,=OSIntCtxSwFlag
        ldr r1,[r0]
        cmp r1,#1
        beq _IntCtxSw                       ; interrupt level context switch

        ldmfd sp!,{r0-r3,r12,lr}
        subs pc,lr,#4                        ; return

        MEND
```

3.2 Timer0 ISR

µC/OS-II clock tick is serviced by calling OSTimeTick() from a timer ISR. Here is timer0 ISR. Duplicating the macro code easily as following, we can get timer0 ISR.

```
;Timer0 interrupt
   IMPORT IRQC_Timer0
IRQASMTimer0 HANDLER IRQC_Timer0
```

IRQASMTimer0 is time0 ISR entry point. IRQC_Timer0 is user code entry point and could be written in C.

Function OSTimeTick() is called by IRQC_Timer0. Most of the work done by function OSTimeTick() basically consist of decrementing OSTCBDly field for each nonzero OS_TCB (task control block). Because OSTCBDly contains the number of clock tick that the task is allowed to delay. OSTimeTick() follows the chain of OS_TCB starting at OSTCBList(list of OS_TCB) until it reaches the idle task. The execution time of OSTimeTick() is directly proportional to the number of tasks created in an application. OSTimeTick() also accumulates the number of clock ticks since power up in an unsigned 32-bit variable called OSTime.

```
void IRQC_Timer0(void) {
     OSTimeTick();                          // serve the clock tick

     TIMER0_IR = 0x01;
     VICVectAddr = 0;                       // clear the interrupt
}
```

4. Time Functions of µC/OS-II

µC/OS-II provides five basic functions to implement time management. They are:

OSTimeDly()

OSTimeDlyHMSM()

OSTimeDlyResume()

OSTimeGet()

OSTimeSet()

OSTimeDly() and OSTimeDlyHMSM() allow the calling task to delay itself for a user specified time. OSTimeDly() calculates the number of ticks to delay and the number can be a value between 1 and 65535. OSTimeDlyHMSM() allow you to specify time in hours, minutes, seconds and milliseconds which is more 'natural'.

OSTimeDlyResume() is used to resume a task that delayed itself. Because there will be another task to cancel the delay and make the delayed task ready-to-run.

When a clock tick occurs, µC/OS II increments a 32-bit counter. At a tick rate of 100Hz, this 32-bit counter rolls over every 497 days. OSTimeGet() can be used to get the value of this counter. You can also change the value of the counter by OSTimeSet().

Before using these functions, you have to give a config in os_cfg.h as following:

```
#define OS_TIME_DLY_HMSM_EN       1       //Include OSTimeDlyHMSM()
#define OS_TIME_DLY_RESUME_EN     1       //Include OSTimeDlyResume()
#define OS_TIME_GET_SET_EN        1        //Include OSTimeGet()and OSTimeSet()
```

Here we give an example on how to implement time management. In the sample application, two tasks are created. TaskMain is used to print out a string and TaskGTime gets OS time and prints it out. By calling function OSTimeDly(), both tasks are delayed 50 clock tick and then go on.

To implement string print out, we use a serial communication interface--Uart port to output some information with which we can easily understand time management of µC/OS-II.

```
#define  STACKSIZE  128

unsigned int TaskMainStack[STACKSIZE];
unsigned int TaskGTimeStack[STACKSIZE];

/*****************************************************************
; Function: SystemInit()
; Parameters: void
; Return: void
; Description: Initialize system according to your application
*****************************************************************/
void SystemInit(void){
   LPC_UART_config_t Uart0_Config;

   //system clock initialization
   TIMER0_InitTimer();

   //Serial port 0 initialization
   Uart0_Config.BaudRate = BD9600;
```

```
   Uart0_Config.WordLenth = WordLength8;
   Uart0_Config.Stopbit=OnebitStop;
   Uart0_Config.ParityEnable = 0;
   Uart0_Config.BreakEnable = 0;
   Uart0_Config.FIFOEnable = 1;
   Uart0_Config.FIFORxTriggerLevel = FIFORXLEV2;
   Uart0_Config.InterruptEnable= IER_RBR  | IER_THRE;     // | IER_THRE ;//| IER_RLS;
   Uart_Init(LPC_UART0, &Uart0_Config);
}

/*********************************************************************
; Function: TaskMain()
; Parameters: void *
; Return: void
; Description: Task TaskMain main body
*********************************************************************/
void TaskMain(void *i){
   SystemInit();               //initialize timer0 and uart0 port
   while(1){
         CommSendString(COMM1,"TaskMain running.\r\n");
         OSTimeDly(50);
    }
}

/*********************************************************************
; Function: TaskGTime()
; Parameters: void *
; Return: void
; Description: Task TaskGTime main body. It will get OS time and display it.
*********************************************************************/
void TaskGTime(void *i){
INT32U tvalue,x;
   char tnumber,narray[15];

   while(1){
         CommSendString(COMM1,"TaskGTime running.\r\n");
         CommSendString(COMM1,"OSTime is:");
         tvalue=OSTimeGet();
         x=0;
         for( ; ; ){
            tnumber=tvalue%10;
            narray[x]=0x30+tnumber;
            tvalue=tvalue/10;
            if(tvalue==0)
break;
            x++;
         }
         for( ; ; ){
            CommPutChar(COMM1, narray[x],0);
            if(x<=0)
break;
```

```
            x--;
        }
        CommSendString(COMM1, "\r\n");
        OSTimeDly(50);
    }
}

/*********************************************************************
; Function: main()
; Parameters: void
; Return: void
; Description: OS initialization, tasl creation and OS start.
*********************************************************************/
int main(void){
   OSInit();

   OSTaskCreate(TaskMain, (void *)0, (OS_STK *)&TaskMainStack[STACKSIZE - 1], 5);
   OSTaskCreate(TaskGTime, (void *)0, (OS_STK *)&TaskGTimeStack[STACKSIZE - 1], 7);
   OSStart();
}
```

In the above sample code, both tasks are delayed 50 clock ticks by calling OSTimeDly(). If you want to specify time in seconds such as one second, you can use function OSTimeDlyHMSM() to rewrite it. For example, TaskMain() can be written as following:

```
void TaskMain(void *i){
   SystemInit();                              //initialize timer0 and uart0 port
   While(1){
           CommSendString(COMM1,"TaskMain running.\r\n");
           OSTimeDlyHMSM (0,0,1,0);
   }
}
```

In order to print the message on PC, a hardware connection shown as figure 2 is needed.

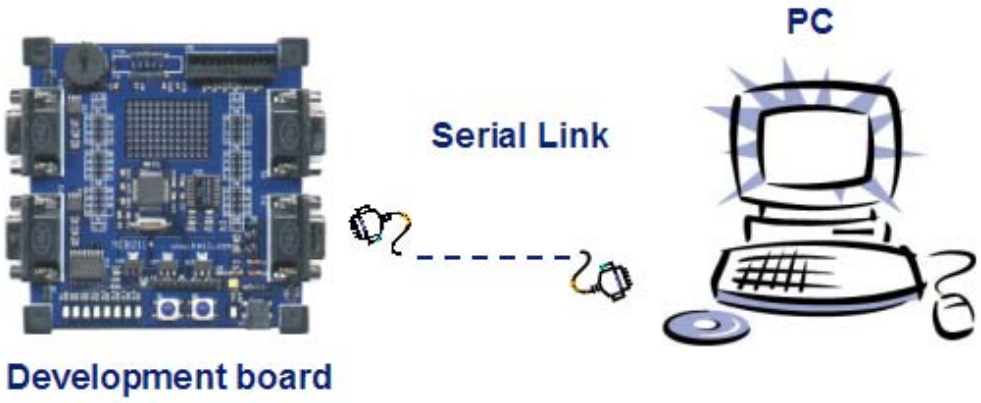

Fig 2. Serial Port Connection

Then we can start HyperTerminal Software on PC. Setting of the software is shown as figure 3.

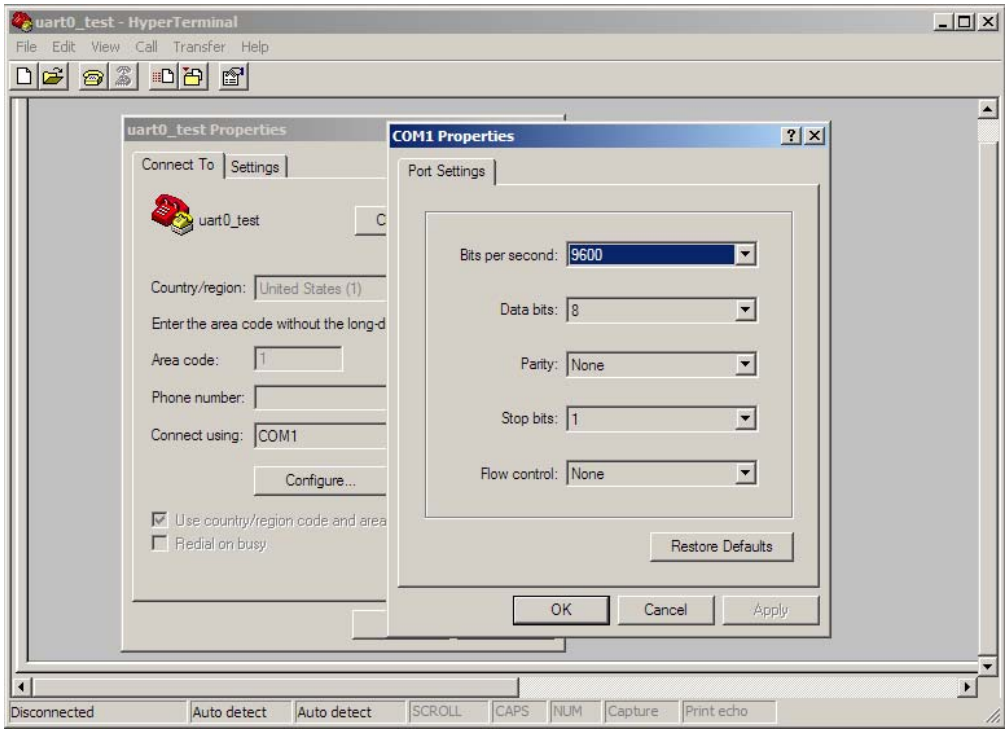

Fig 3. Setting of HyperTerminal Software

After these configurations, run the code. Figure 4 shows the printed messages. Because TaskMain has higher priority than TaskGTime, TaskMain runs first and print out ”TaskMain running.”. Calling OSTimeDly() causes TaskMain to delay itself for 50 clock ticks. A context switch occurs. TaskMain is pending and TaskGTime, the next highest priority ready-to-run task, starts to run. It prints out ”TaskGTime running.” and OS time. OSTimeDly() also delay TaskGTime for 50 clock ticks. So from the printed messages, we can see that both tasks run alternately. And the printed OS time is increased by 50 equal to delay time.

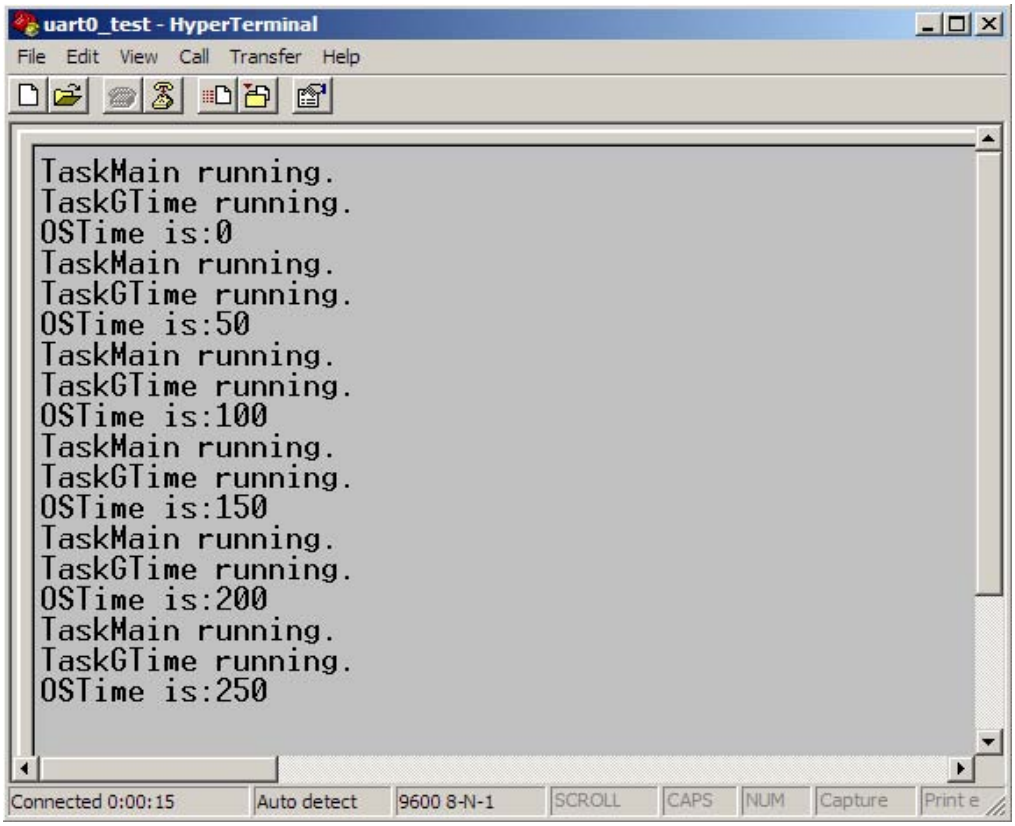

Fig 4. Printed Messages

5. Disclaimers

Life support — These products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such application.

Right to make changes — Philips Semiconductors reserves the right to make changes in the products - including circuits, standard cells, and/or software - described or contained herein in order to improve design and/or performance. When the product is in full production (status 'Production'), relevant changes will be communicated via a Customer Product/Process Change Notification (CPCN). Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no licence or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

Application information — Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors make no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

6. Trademarks

Notice — All referenced brands, product names, service names and trademarks are the property of the respective owners.

7. Contents

Date of release:15 December 2005
Document number: AN10413_1

Published in The Netherlands