AN10413

µC/OS-II time management in LPC2000

Rev. 02 — 18 July 2007

Application note

Document information

Info	Content
Keywords	uC/OS-II, MCU, ARM, LPC2000, Timer, IRQ, VIC
Abstract	This application note demonstrates how to implement µC/OS-II time management in the LPC2000 microcontroller family from NXP Semiconductors. This application note also serves as a quick-start guide and includes a simple time management code example.

Revision history

Rev	Date	Description
02	20070718	• The format of this application note has been redesigned to comply with the new identity guidelines of NXP Semiconductors. • Legal texts have been adapted to the new company name where appropriate.
01	20051215	First release

Contact information

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

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

The μC/OS-II, pronounced ‘Micro C O S 2’, and stands for MicroController Operating System version 2, is a type of real-time operating system. Its real-time kernel, easy port connection, and reliability enable it to be used in a wide variety of applications, such as cameras, medical instruments, engine controls, and ATMs. The μC/OS-II can run on most 8/16/32-bit microprocessors or microcontrollers.

An important function of the μC/OS-II is time management. It provides periodic interrupts for the purpose of keeping track of time delays and time-outs. An interrupt period is called a Clock Tick which represents the system’s heartbeat. Usually, a Clock Tick (tick) should occur between 10 and 100 times per second (Hz). The faster the tick rate, the higher the overhead imposed on the system. The actual frequency of the tick depends on the tick resolution required by the user application. Usually the tick source is provided by a hardware timer.

The LPC2000 family is based on the 16/32-bit ARM7TDMI-S microcontroller. The μC/OS-II is supported by all the devices in the LPC2000 family including two 32-bit timer/counter devices which can be used as a Clock Tick source. In this application note we use Timer0 as an example, and Timer0 will be configured to periodically trigger an IRQ interrupt. The code is developed in ARM Development Suite (ADS) v1.2 and written mostly in ANSI C. The code was tested on an evaluation board with an LPC2129, which uses a 12 MHz crystal.

2. Initialization

2.1 Exception vector table

The ARM CPU contains an exception vector table supporting seven types of exception. When an exception occurs, an execution is forced from fixed memory whose address corresponds to the exception type. The exception vector table for the ARM is shown in Table 1.

Table 1. Exception vector table

Exception	Mode	Vector address
Reset	SVC	0x0000 0000
Undefined Instruction	UND	0x0000 0004
Software Interrupt (SWI)	SVC	0x0000 0008
Prefetch abort	Abort	0x0000 000c
Data abort	Abort	0x0000 0010
-	-	0x0000 0014
IRQ (normal interrupt)	IRQ	0x0000 0018
FIQ (fast interrupt)	FIQ	0x0000 001c

On reset, the CPU begins executing from the reset vector entry, then jumps to an initialization sub-routine, which starts 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 sub-routine
        LDMFD    SP!, {R0-R3, LR}
        SUBS     PC, LR, #4
    END
```

Note that the handlers shown in the startup code do not do anything useful. They are only shown here for completeness. You can implement them according to your application.

2.2 System configuration

System configuration 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 the CPU run at the full speed of 60 MHz, PLL is set to 5. And the VPB is set to a quarter of the CPU speed. Using the Memory Map Register, you can remap interrupt vectors from 0x0000 0000 to 0x000 0001c (on-chip flash), 0x4000 0000 to 0x4000 001c (on-chip RAM) or 0x8000 0000 to 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 the Clock Tick. The tick frequency is defined as **OS_TICKS_PER_SEC** in file os_cfg.h. Timer0 counter is set according to the frequencies of the Clock Tick and the peripheral clock.

The LPC2000 family contains a VIC that supplies a vector (address) for each interrupt source. The 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 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. Table 2 lists the interrupt sources for each peripheral function.

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

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

```
LDR pc, [pc,#-0xFF0]
```

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

Table 2. Connection of interrupt sources to VIC

Block	Flag	VIC channel
WDT	Watchdog Interrupt (WDINT)	0
-	reserved for software interrupts only	1
ARM core	embedded ICE, DbgCommRx	2
	embedded ICE, DbgCommTx	3
TIMER0	Match 0 to 3 (MR0, MR1, MR2, MR3)	4
	Capture 0 to 3 (CR0, CR1, CR3)	
TIMER1	Match 0 to 3 (MR0, MR1, MR2, MR3)	5
	Capture 0 to 3 (CR0, CR1, CR3)	

Table 2. Connection of interrupt sources to VIC *…continued*

Block	Flag	VIC channel
UART0	Rx Line Status (RLS)	6
	Transmit Holding Register Empty (THRE)	
	Rx Data Available (RDA)	
	Character Time-out Indicator (CTI)	
UART1	Rx Line Status (RLS)	7
	Transmit Holding Register Empty (THRE)	
	Rx Data Available (RDA)	
	Character Time-out Indicator (CTI)	
	Modem Status Interrupt (MSI)	
PWM0	Match 0 to 6 (MR0, MR1, MR2, MR3, MR4, MR5, MR6)	8
I^{2}C	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
	External Interrupt 1 (EINT1)	15
	External Interrupt 2 (EINT2)	16
	External Interrupt 2 (EINT2)	17
A/D	A/D Converter	18
CAN	CAN and Acceptance Filter:	
	1 ORed CAN, LUTerr int	19
	CAN1 and CAN2: 2 × (Tx int, Rx int) LPC2119/2129/2292/2294	20 to 23
	CAN3 and CAN4: 2 × (Tx int, Rx int) LPC2194/2292/2294 only	24 to 27

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

```
#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
```

```
}
```

Note that in µC/OS-II, you must enable Clock Tick interrupts after multi-tasking 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 tick interrupts after calling OSInit() and before OSStart() as shown in the following code, because at that point the µC/OS-II is in an unknown state and your application will crash.

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

3. Clock tick ISR

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

It is possible that after the last nested ISR has completed, an interrupted task is not required to run because a new higher priority task has occurred. This is handled by an interrupt level context switch, implemented by function _IntCtxSw(), so that after return, the new higher priority task runs while the old lower priority task is kept pending.

Write ISR codes in assembly language because CPU registers cannot be accessed directly with C code; however, user code can be written in C. In the following example, macro code is used to implement an ISR in file irq_handler.s. The code can be as shown and should be copied 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 Sub-routine
      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
```

3.1 Timer0 ISR

The µC/OS-II Clock Tick is serviced by calling OSTimeTick() from a timer ISR. In the following example it is Timer0 ISR. Copying the macro code as shown gives Timer0 ISR.

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

IRQASMTimer0 is Timer0 ISR entry point. IRQC_Timer0 is the user code entry point and can be written in C.

Function OSTimeTick() is called by IRQC_Timer0. Most of the work done by function OSTimeTick() basically consists of decrementing field OSTCBDly for each non-zero OS_TCB (Task Control Block). Because OSTCBDly contains the number of Clock Ticks 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

The µC/OS-II provides five basic functions for implementing 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: a value between 1 and 65535. OSTimeDlyHMSM() allows you to specify time in hours, minutes, seconds and milliseconds which is more ‘natural’.

OSTimeDlyResume() is used to resume a task that delayed itself. 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 100 Hz, 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 configuration in os_cfg.h as follows:

```
#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 is an example of 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 for 50 Clock Ticks before continuing.

To implement string print-out, a serial communication interface UART port is used to output some information with which time management of the μC/OS-II can be easily understood.

```
#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, task 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 for 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 follows:

```
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 a PC, a hardware connection is required as shown in Figure 1.

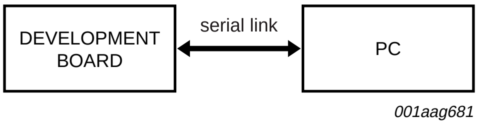
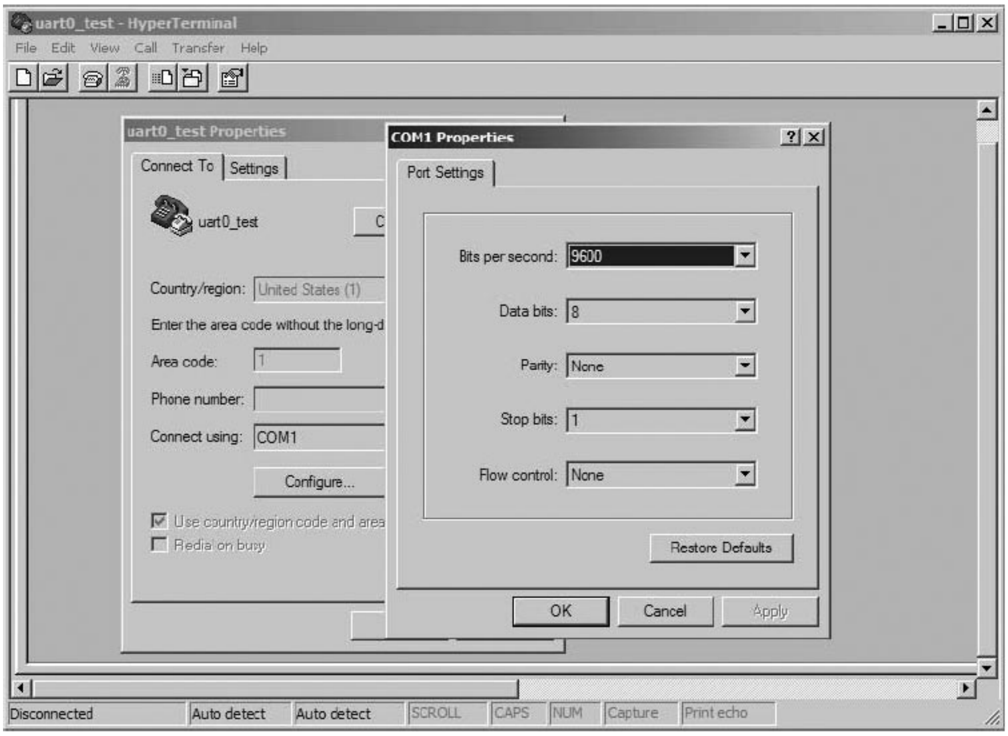


Fig 1. Serial port connection

HyperTerminal Software on the PC can now be started. Setting of the software is shown in Figure 2.



uart0_test

Country/region: United States (1)

Enter the area code without the long-d

Area code: 1

Phone number:

Connect using: COM1

☑ Use country/region code and area

☐ Redial on busy

Bits per second: 9600

Data bits: 8

Parity: None

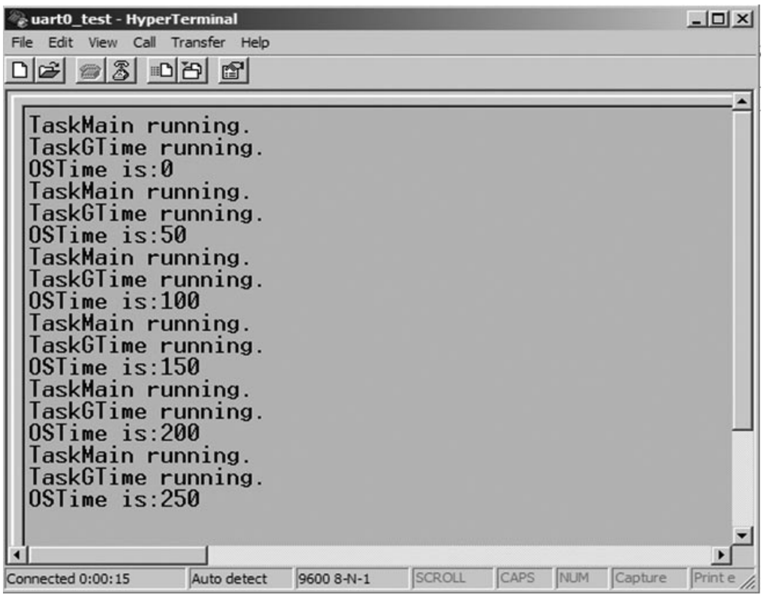
Stop bits: 1

Flow control: None

Fig 2. Setting of HyperTerminal Software

Now the system has been configured, run the code on the LPC2xxx. Figure 3 shows the printed messages.

Because TaskMain has a higher priority than TaskGTime, TaskMain runs first and prints 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 the OS time. OSTimeDly() also delays TaskGTime for 50 Clock Ticks. So from the printed messages, we can see that both tasks run alternately. The printed OS time is increased by 50 Clock Ticks which is equal to the programmed delay time.



```
TaskMain running.
TaskGTime running.
OSTime is:0
TaskMain running.
TaskGTime running.
OSTime is:50
TaskMain running.
TaskGTime running.
OSTime is:100
TaskMain running.
TaskGTime running.
OSTime is:150
TaskMain running.
TaskGTime running.
OSTime is:200
TaskMain running.
TaskGTime running.
OSTime is:250
```

001aag683

Fig 3. Printed messages

5. Abbreviations

Table 3. Abbreviations

Acronym	Description
ARM	Advanced RISC Machine
ATM	Automated Teller Machine
FIQ	Fast Interrupt Request
ISR	Interrupt Service Request
MAM	Memory Accelerator Module
MCU	MicroController Unit
SVC	Supervisor
UART	Universal Asynchronous Receiver Transmitter
UND	Undefined
VIC	Vectored Interrupt Controller
VPB	VLSI Peripheral Bus
VPBDIV	VLSI Peripheral Bus Divider

6. Legal information

6.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

6.2 Disclaimers

General — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of a NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

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

6.3 Trademarks

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

© NXP B.V. 2007. All rights reserved.

7. Contents

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 18 July 2007
Document identifier: AN10413_2