AN10548

Getting started with LPC288x

Rev. 01 — 8 January 2007 **Application note**

Document information

Info	Content
Keywords	LPC2880, LPC2888, Flash, Interrupt handling, Timer, CGU
Abstract	This application note provides code examples for the various peripherals of the LPC288x. This application note also makes an attempt to draw a comparison of the LPC288x with the other devices of the LPC2000 family

Revision history

Rev	Date	Description
01	20070108	Initial version

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 LPC288x (LPC2880/LPC2888) includes an ARM7TDMI CPU with an 8 kB cache. The LPC2888 includes a 1 MB Flash memory system. It has an on-chip DC-to-DC converter that can generate the required voltages from a single battery or from USB power. Some other important peripherals are a high-speed USB 2.0 device interface, I^2S interface, SD/MMC card interface and 16-bit stereo A/D and D/A converters (with amplification and gain control).

In many ways, the LPC2888/LPC2880 is different compared to other devices in the LPC2000 family. The LPC288x differs in the clocking structure; interrupt handling and flash programming (to list a few). So code written for an LPC2000 device may not necessarily work for the LPC288x without modifications.

This application note attempts to cover the various aspects that need to be considered while writing an application for the LPC288x or porting existing software from any LPC2000 family device to the LPC288x.

The various topics covered in this application note are as follows:

1. Basic assembly startup code
2. Port pins
3. Clock Generation Unit (CGU)
4. Interrupt handling using Cache and Timer0
5. Flash programming

2. Basic startup code

The startup assembly code for the LPC288x is similar to the other LPC2000 devices since it is also based on the ARM7TDMI-S architecture. One has to of course take into consideration that the on-chip Flash and the on-chip SRAM start from address 0x1040 0000 and 0x40 0000, respectively, so the stack pointers and heap should be arranged accordingly. In other LPC2000 devices, the Flash begins from 0x0 and the SRAM starts from 0x4000 0000.

There are seventeen locations in the Clock Generation Unit (CGU) module which need to be initialized to 0 to reduce the overall power consumption. In the header file, these locations have been addressed as X1 to X17.

In a typical LPC2000 device startup file, one might also find code for initializing the PLL and the Memory Accelerator module (MAM). This cannot be used for the LPC288x since the clocking structure is more involved which includes seven input clocks and two different PLLs. Also there is no MAM in the LPC288x.

3. Port pins

The LPC288x has 89 GPIO pins that are split into 8 ports (Port0 to Port7). Some major differences in the port structure are as follows:

1. In other LPC2000 devices all the port pins were initialized to their GPIO configuration on reset. In LPC288x, four port pins come up as GPIO and the rest are initialized to their Functional IO settings. The four pins that are configured to GPIO belong to Port2.

In LPC2000, to toggle port pins one has to set the direction of the pin (IODIRx) as output and use IOSETx and IOCLRx respectively to drive the pin high and low.In LPC288x, there are two-register bits m1 and m2, which need to be set to do the same operation. The table below shows 2 register bits and their corresponding impact on port pins:

Table 1. M1/M0 settings

m1	m1	Pin state
0	0	GP in (not driven)
0	1	Functional I/O
1	0	GP out (drive LOW)
1	1	GP out (drive HIGH)

As shown above, m1 has to be set and if we toggle the m0 bit, then the corresponding port pins will toggle.

3.1 Code example

Below is a code snippet showing how the 4 port pins of Port2 can be toggled in software.

```
1               /* Setting Port2 as “GP out (drive low)” */
2               MODE1S-2=0xF;
3               MODE0C-2=0x0;
4
5               /* To toggle the pins, just set and reset bit m2  */
6               while(1)
7                    {
8                       MODE0S-2=0xF;
9                          MODE0C-2=0xF;
10                   }
```

4. Clock Generation Unit (using UART)

The CGU is the most important block of the LPC288x and it controls the clocks for the different modules of this device. The CGU is not present in any of the other LPC2000 devices.

4.1 Basic configuration

For a basic configuration of the CGU the following steps need to be carried out:

1. PLL registers: *(optional)*
   a. Configure either the Main PLL or the high speed PLL if higher speeds are expected for the application.

2. Selection stage registers:
   a. Use the respective Frequency Select register to select one of the 7 input clocks.
   b. The Switch Configuration and Switch Status registers would be typically used while dynamic clock switching (shown below with an example).
3. Spreading stage registers: *(optional)*
   a. Power control/Power status/ Enable Select and Software Reset registers would not be typically used for a basic CGU configuration.

An application is provided below that shows how to configure the CGU for the UART peripheral. The UART is initially set to work with the 12 MHz oscillator input and then the clock source is dynamically and gracefully switched between the 32.768 KHz oscillator input and the output of the main PLL (60 MHz). This application is built for the Nohau evaluation board, details of which can be found by browsing to the “Support & tools” section of:

http://www.nxp.com/pip/LPC2880FET180.html#support

The board has a UART port, which should be connected to the PC serial port using a Null modem cable. The output is captured on Tera Term Pro v2.3 (similar to HyperTerminal) and it is set to run at the serial baud rate of 600 baud.

4.2 Code example

```
1
2
3    /* Header file for LPC288X */
4    #include"LPC288x.h"
5
6    /* Used by UART */
7    #define TEMT (1<<6)
8    #define LINE_FEED 0xA
9    #define CARRIAGE_RET 0xD
10
11   void CGU(void);
12   void switch_clk(int);
13   void printP(char[],int);
14
15
16
17   /*****************************************************************
18    * Configuring the CGU
19    *****************************************************************
20    */
21   void CGU()
22        {
23             /* Enabling the main PLL. Output of PLL is 60MHz */
24             LPFIN=0x1;       /* Select the clock source as the 12MHz crystal */
25             LPMSEL=4;        /* Multiplication Factor */
26             LPPSEL=1;        /* Division Factor */
27             LPPDN=0;         /* Maintaining the reset condition- enabled */
28
29             while(!(LPLOCK & 1)){}
30
```

```
31              /* Selection stage- UART */
32              UARTSCR=0x1;     /* Enables Side 1 of selection stage */
33              UARTFSR1=0x1;    /* Select the clock source as the 12MHz crystal */
34
35          }
36
37    /*************************************************************
38     * Switching clocks between 60MHz and 32.768KHz osc. input
39     * Parameter 's' has a defined value.
40     * s=1-> Main PLL
41     * s=2-> 32.768KHz
42     *************************************************************
43     */
44    void switch_clk(int s)
45          {
46              int temp;
47              int new_clk;
48
49              /* If s=1 then configure the UART for Main PLL else for the 32.768KHz crystal*/
50              if(s==1)
51                   {
52                        /* Set the new clock as the main PLL source. This value is used
53                         * later while setting the Frequency select register
54                         */
55                        new_clk=0x8;
56
57                        // Configuring UART for Main PLL
58                        UARTRES=0x0;
59                        UARTRES=0x1;
60
61                        U0_FCR = 0x7;
62                        U0_LCR = 0x83;
63                        U0_DLL = 0x6a;        /* Divider set to 6250 */
64                        U0_DLM = 0x18;
65                        U0_LCR = 0x3;
66                   }
67              else
68                   {
69                        /* Set the new clock as the 32.768KHz. This value is used
70                         * later while setting the Frequency select register
71                         */
72                        new_clk=0x0;
73
74                        // Configuring UART for 32.768KHz osc
75                        UARTRES=0x0;
76                        UARTRES=0x1;
77
78                        U0_FCR = 0x7;
79                        U0_FDR=  0xF2;
80                        U0_LCR = 0x83;
81                        U0_DLL = 0x3;              /* Divider set to 3 */
```

```
82                        U0_LCR = 0x3;
83                  }
84
85            /* Read the Switch status register to check
86             * which side of the stage is enabled
87             */
88            temp=UARTSSR;
89
90            if(temp & 1)
91                  {
92                        /* Set the clock on Side 2 */
93                        UARTFSR2=new_clk;
94                        /* Write to the Switch Configuration register */
95                        UARTSCR=(temp & 0x3)^ 0x3;
96                        /* Wait till the switch is complete*/
97                        while(!(temp & 0x2))
98                              {
99                                    temp=UARTSSR;
100                             }
101                 }
102           else
103                 {
104                       /* Set the clock on Side 1 */
105                       UARTFSR1=new_clk;
106                       /* Write to the Switch Configuration register */
107                       UARTSCR=(temp & 0x3)^ 0x3;
108                       /* Wait till the switch is complete */
109                       while(!(temp & 0x1))
110                             {
111                                   temp=UARTSSR;
112                             }
113
114                  }
115       }
116
117 /*************************************************************
118  * Function for printing characters on the Terminal program (PC).
119  * The function takes two parameters, a character array "a" and an integer "k"
120  * It prints the character array "a", "k"times
121  *************************************************************
122  */
123 void printP(char a[],int k)
124       {
125           int i=0,j=0;
126
127           for(j=0;j<k;j++)
128                 {
129                       while(a[i])
130                             {
131                                   U0_THR=a[i];
132                                   while(!(U0_LSR & TEMT)){}
```

```
133                                 i++;
134                           }
135                           U0_THR=CARRIAGE_RET;
136                           U0_THR=LINE_FEED;
137                           while(!(U0_LSR & TEMT)){}
138
139                      i=0;
140                 }
141
142
143       }
144
145  /*****************************************************************
146   *                                  MAIN
147   *****************************************************************
148   */
149  int main(void)
150       {
151            /* Characters arrays */
152            char lpc[]="LPC288x";
153            char a[]="++++++++++++++++++++++++++++++++++++";
154            char b[]="------------------------------------";
155            char uart_12[]="UART Running from 12MHz Oscillator";
156            char uart_pll[]="UART Running from Main PLL";
157            char uart_32[]="UART Running from 32.768KHz";
158
159
160            /* Configuring the CGU */
161            CGU();
162
163            /* UART Configuration to derive 600baud from the 12MHz main oscillator */
164            U0_FCR = 0x7;
165            U0_LCR = 0x83;
166            U0_DLL = 0xE2;             /* Divider set to 1250 */
167            U0_DLM = 0x4;
168            U0_LCR = 0x03;
169
170            /* Printing characters */
171            printP(a,1);
172            printP(uart_12,1);
173            printP(lpc,5);
174            printP(b,1);
175
176            /* Continous switching is done between 60MHz and
177             * 32.768Khz
178             */
179            while(1)
180                 {
181                      /* Switching to Main PLL and configuring the UART dividers for the same*/
182                      switch_clk(1);
183                      /* Printing characters */
```

```
184                     printP(uart_pll,1);
185                     printP(lpc,5);
186                     printP(b,1);
187
188                     /* Switching to 32.768KHz and configuring the UART dividers for the same */
189                     switch_clk(2);
190                     /* Printing characters */
191                     printP(uart_32,1);
192                     printP(lpc,5);
193                     printP(b,1);
194                 }
195
196         }
```

See Fig 1 for sample output.

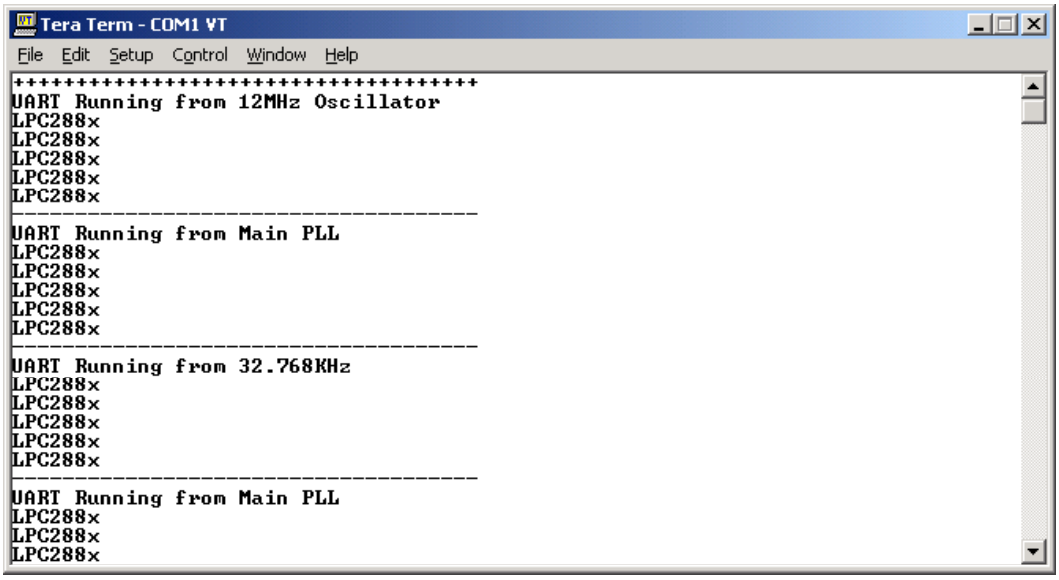

Fig 1. Sample output

5. Interrupt handling (using the cache and Timer0)

Interrupt handling in the other LPC2000 devices was handled by the Vectored Interrupt Controller (VIC) and it is different than the interrupt controller in the LPC288x. The main difference between the two controllers is that there was a provision in the VIC to insert an Interrupt Vector Table (IVT) within the VIC itself. The IVT in the LPC288x has to be stored in memory.

5.1 Basic configuration

The interrupt controller in the LPC288x requires at least two registers to be configured, which would be as follows:

1. Interrupt Request register for a particular interrupt source:
   a. There is an Interrupt request register for each interrupt source and here, one can mainly set the priority level of the interrupt source, assign the source as an IRQ/FIQ interrupt and also enable the interrupt.
2. Vector register (INT_VECTOR0 for IRQ and INT_VECTOR1 for FIQ):
   a. In the vector register, the base address of the IVT has to be programmed. This register also has index bits, which get updated when an IRQ/FIQ interrupt is triggered. The vector register for IRQ has the following structure as shown in Fig 2:

INT_VECTOR0		
Bits	**Name**	**Description**
31:11	Table_addr	Base address of table
7:3	Index	Hardware would set these bits to the appropriate interrupting source

Fig 2. IRQ vector register

5.2 Multiple interrupt sources for a single peripheral

This interrupt controller can also accept multiple inputs from the same interrupt source, which enables priority levels within a specific peripheral. For instance, the USB has four inputs to the interrupt controller enabling four levels of interrupt priority within USB interrupts. Fig 3 shows a snapshot of the user manual, which illustrates this feature:

Table 107. LPC288x interrupt sources

Bit # / register #	Block	Source
23	Flash Programming	Programming or Erasure Complete
24	LCD Interface	LCD FIFO Empty
		LCD FIFO Half Empty
		LCD FIFO Overrun
		LCD Read Valid
25	GPDMA	
26-29	USB	USB Frame
		Endpoint 0 - 7
		Device Status
		Command Code Empty
		Command Data Full
		EOP Reached for Out Transfer
		EOP Reached for In Transfer

Fig 3. Interrupt sources

5.3 Cache and Interrupt Vector Table (IVT) configuration

While programming interrupts for the LPC288x the following should be considered:

1. Cache configuration: There is no memory residing at 0x0. In other LPC2000 devices, the on-chip flash always existed from 0x0. Since on an exception the ARM7 core jumps to one of the locations between 0x0 to 0x1C, memory has to be remapped to this location to handle the same. In the LPC288x memory can be remapped to 0x0 using the cache. Hence the cache has to be setup to handle exceptions.
2. Interrupt vector table (IVT): The interrupt controller cannot accept individual ISR addresses for each interrupt source. Only the base address of the IVT can be programmed, which would reside somewhere in memory. A typical IVT would have the following structure:

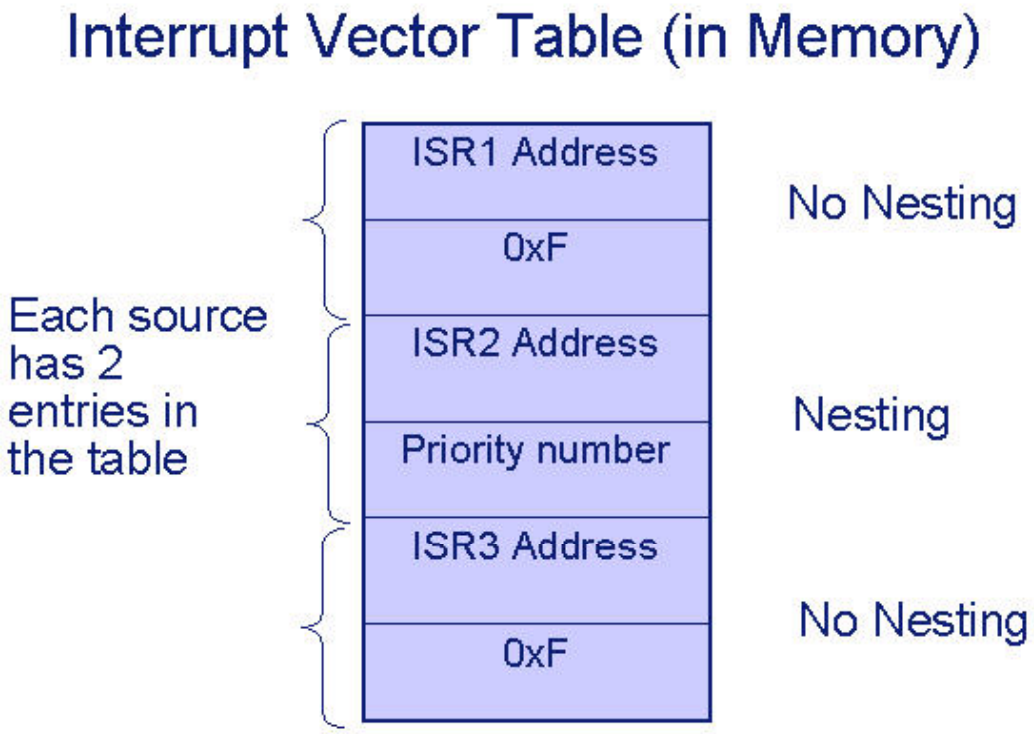

Fig 4. Interrupt vector table

As shown in Fig 4, each interrupt source has 2 entries in the IVT. The first entry is the address of the ISR and the second entry has the priority level of the interrupt source that can interrupt this particular interrupt. If interrupt nesting were not desired then the second entry in the IVT would have a value of “0xF”. For a complete understanding on nested interrupt handling, please refer to the “Interrupt Controller Usage Notes” section in the LPC288x User manual.

5.4 Simple interrupt handling

Lets consider IRQ interrupt handling in the LPC288x. As part of the initialization, the following needs to be done:

a. The basic startup code with the interrupt vectors and the SP for IRQ has been set in software.

b. Cache would be configured so that memory resides at 0x0.

c. Interrupt Request register has been set

d. INT_VECTOR0 register has been set with the base address of the IVT.

e. The IVT is configured in software.

When the interrupt fires, the following sequence of events can be expected:

1. The index bits in the INT_VECTOR0 register gets updated with the interrupt source number and this also creates the ISR address of this interrupt source

2. The ARM7 core would execute the instruction that resides at 0x18, which would be a jump to a top-level handler that would read the INT_VECTOR0 register and then jump to the ISR.
3. In the ISR, the interrupt will be serviced and the corresponding interrupt will be cleared in the peripheral.

The sequence of events that happen after the interrupt is triggered (not considering nesting) is shown in Fig 5:

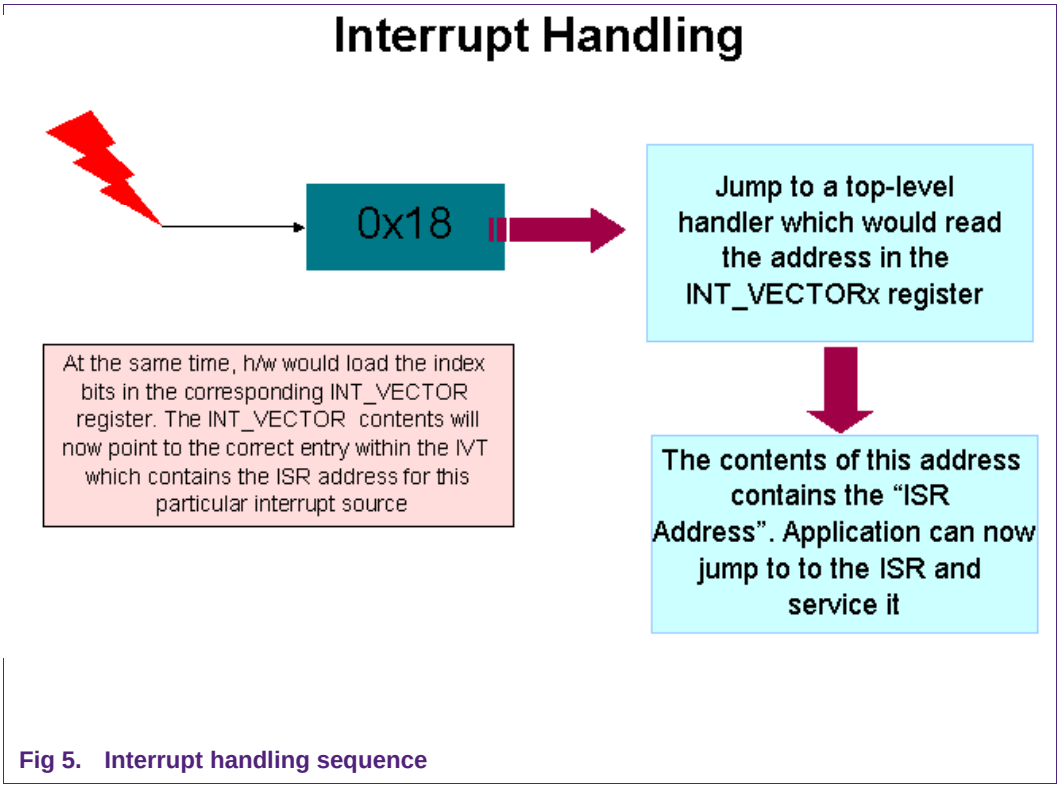

Fig 5. Interrupt handling sequence

5.5 Code example

An application is provided below which is built for the Nohau LPC2888 evaluation board. The application configures Timer0 as an IRQ interrupt and interrupts the core every 10 seconds. An LED is provided on the board, which blinks slowly while the code is in the main loop and blinks at a much faster rate while it is in the ISR.

The code is linked to run from the internal SRAM and the cache is used to remap memory from the SRAM region to 0x0.

5.5.1 Startup assembly code

The exception vectors that would ultimately be mapped to 0x0 from SRAM are shown below. These assembly instructions were compiled under the ARM ADS environment.

```
1                AREA IVT, CODE
2                CODE32
3                IMPORT reset
4                IMPORT IRQHandler
5
```

```
6                   ENTRY
7            ; First instruction to be executed which would jump to a
8            ; section of assembly code with entry point as “reset”. Here
9            ; the SP is set for the IRQ mode and the Supervisor mode.
10           LDR     PC, =reset
11           LDR     PC, Undefined_Addr
12           LDR     PC, SWI_Addr
13           LDR     PC, Prefetch_Addr
14           LDR     PC, Abort_Addr
15           NOP                             ; Reserved vector
16   ; This is the IRQ vector. On an IRQ interrupt, the ARM7 core
17   ; would execute the below instruction and jump to “IRQHandler”
18   ;  which is in the C Code.
19           LDR     PC,=IRQHandler
20           LDR     PC, FIQ_Addr
21
22
23   Undefined_Addr  DCD     Undefined_Handler
24   SWI_Addr        DCD     SWI_Handler
25   Prefetch_Addr   DCD     Prefetch_Handler
26   Abort_Addr      DCD     Abort_Handler
27   FIQ_Addr        DCD     FIQ_Handler
28
29
30   ; ************************
31   ; Exception Handlers
32   ; ************************
33
34   ; The following dummy handlers do not do anything useful in this example.
35   ; They are set up here for completeness.
36
37   Undefined_Handler
38           B       Undefined_Handler
39   SWI_Handler
40           B       SWI_Handler
41   Prefetch_Handler
42           B       Prefetch_Handler
43   Abort_Handler
44           B       Abort_Handler
45   FIQ_Handler
46           B       FIQ_Handler
47
48           END
49
50
```

5.5.2 C code

```
1    /* Header file and other macro definitions*/
2    #include"LPC288x.h"
3    #define MASK_INDEX         0xFFFFF800      /* Mask bits for Index bits in the INT_VECTOR0 register*/
4    #define TABLE_BASE         0x40D000        /* Base address of IVT */
5    #define TIMER_10SECS  0x7441E
6    #define SLOW               1               /* Used by the LED function */
7    #define FAST               2
8
9    /* Function Declarations */
10   void led_blink(int);
11   extern __irq void IRQHandler(void);
12   void IRQ_timer(void);
13   void install_handler(int,void (* )(),int);
14
15   /*******************************************************
16    * Toggles the LED on Nohau board
17    *******************************************************
18    */
19   void led_blink(int a)
20        {
21             int j,m;
22
23
24             /* Pin Toggling code for the Four GPIO pins */
25             MODE1S_2=0xF;
26             MODE0C_2=0x0;
27
28             if(a==1)
29                  {
30                       /* Slow toggle */
31                       for(j=0;j<3;j++)
32                            {
33
34                                 MODE0C_2=0xF;
35                                 for(m=0;m<100000;m++){}
36
37                                 MODE0S_2=0xF;
38                                 for(m=0;m<100000;m++){}
39                            }
40                  }
41
42             if(a==2)
43                  {
44                       /* Fast toggle */
45                       for(j=0;j<50;j++)
46                            {
47
48                                 MODE0C_2=0xF;
49                                 for(m=0;m<10000;m++){}
50
```

```
51                                  MODE0S_2=0xF;
52                                  for(m=0;m<10000;m++){}
53                              }
54                  }
55
56
57          }
58
59    /****************** Top Level ISR ************************
60     * This function will be called from 0x18 (IRQ vector).
61     * Here, the INT_VECTOR0 register would be read, which will
62     * have the ISR address of Timer0
63     *************************************************************
64     */
65    __irq void IRQHandler()
66          {
67                void (* fptr )();
68                unsigned int *temp;
69
70                /* Read the INT_VECTOR0 register and store the ISR address in "temp" variable */
71                temp=(unsigned int *)INT_VECTOR0;
72                /* fptr would now point to that ISR location */
73                fptr=(void (*)(void))(* temp);
74
75                /* Call IRQ_Timer() */
76                fptr();
77
78          }
79
80
81
82    /****************** Timer ISR*****************************
83     * Handle the Timer interrupt
84     *************************************************************
85     */
86    void IRQ_timer()
87          {
88                /* Toggle port pin */
89                led_blink(FAST);
90
91                /* Clear the timer interrupt */
92                T0CLR=0x0;
93          }
94
95    /****************** ISR Table address ********************
96     * Function that would install the ISR address
97     * (pointed by function pointer "f")
98     * and the priorty number "b" in the IVT with
99     * base address 0x40D000.The interrupt source number "a"
100    * is used to vector to the correct offset from the
101    * base address of the IVT
```

```
102   ******************************************************
103  */
104  void install_handler(int ind,void (* f )(), int pr)
105       {
106            unsigned int * temp;
107
108            /* Use the Interrupt source number to index to the ISR address entry from base address*/
109            temp=(unsigned int *)(TABLE_BASE |(ind<<3));
110
111            /* Load the ISR address and the priorty number in the IVT*/
112            *temp=(unsigned int)f;
113            temp++;
114            *temp=pr;
115       }
116
117
118  /*********************************************************
119   *                              MAIN
120   *********************************************************
121   */
122  int main()
123       {
124            /* Function pointer which is used to install the ISR
125             * address and the priorty number into the IVT */
126            void (*func_ptr)();
127
128            /* Initalizing Cache Controller */
129            CACHE_SETTINGS=0x1;                       /* Reset the cache */
130            CACHE_SETTINGS=0x0;                       /* De-assert reset to the cache controller */
131            while((CACHE_RST_STAT) & 0x1){}       /* Wait for reset to complete */
132            CACHE_PAGE_CTRL=0x1;                  /* Enable virtual page 0 */
133            ADDRESS_PAGE_0=(0x400000 >>21);  /* Prepare virtual address */
134            CACHE_SETTINGS=0x16;                  /* Enable caching */
135
136            /* Configure Interrupt Controller */
137            INT_REQ5=((1<<27)|(1<<26)|(1<<16)|(1<<28)|0x1);
138            /* The priorty of the interrupt has to be set along with the WE_PRIO bit */
139            func_ptr=IRQ_timer;  /* func_ptr now points to the function
140                                    IRQ_timer */
141            install_handler(5,*func_ptr,15);
142            INT_VECTOR0=TABLE_BASE & MASK_INDEX;
143
144            /* Configure Timer0 */
145            T0LOAD=TIMER_10SECS;  /* Timer would be interrupting every 10seconds  */
146            T0CTRL=0xC8;
147            /* Timer Counter is set at CGU/256 and the load option is selected  */
148
149            /* Blink LED slowly */
150            while(1)
151                 {
152                      led_blink(SLOW);
```

```
153                 }
154
155         }
```

6. Flash programming

The LPC2888 has a 1 MB on-chip Flash system, which can be programmed either via the USB interface or via the user application itself. Both these interfaces are different considering the other LPC2000 devices. LPC2000 devices can be programmed via the UART or via the bootloader IAP interface.

A User Manual describing Flash programming via USB is included in the “LPC2888 Flash Programming Utility” zip package (which is available for download). The bootloader on the LPC2888 does not provide an interface for the Flash programming. Instead the Flash memory controller registers are directly provided in the LPC288x User Manual. Flash programming in the LPC2888 is achieved by programming small units within a sector called “page” (which is of size 512 bytes). The concept of a page is not present in other LPC2000 devices.

6.1 Flash programming flowchart

Fig 6 shows a flowchart outlining the flash programming steps in the LPC2888:

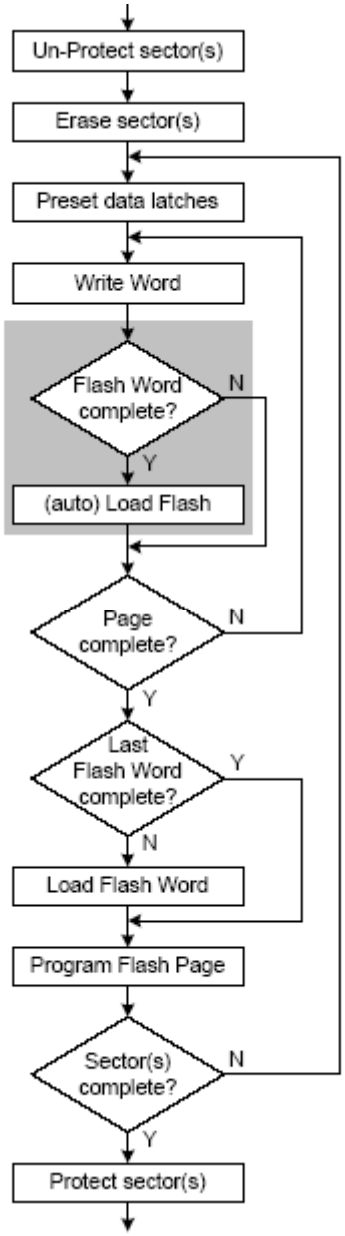

Fig 6. Flash programming steps

The details on how to use the flash memory controller registers are explained in the LPC288x User Manual.

6.2 Signature for valid user code

In LPC2000 devices, in an attempt to execute user code, the on-chip bootloader disables the overlaying of the interrupt vectors from the boot block (on reset), then checksums the interrupt vectors in Sector 0 of the flash. If the checksum matches the 2's complement of the interrupt vectors (this computation would exclude the word located 0x14) which resides at 0x14 then the execution control is transferred to user code by loading the program counter with 0x0.

In LPC2888, if port pins P2.2 and P2.3 are left unconnected then the device enters Mode 0, which would be an attempt to execute user code from the internal flash. The bootloader would then check for a signature, which would reside at 0x104F F800. This signature or marker would be 0xAA55 AA55. If this signature is not found at the specified location then the mode is switched to mode2, which is the USB download mode.

6.3 C code

Below, C code is provided which shows how a single sector (Sector 0) can be programmed. The same code example can be expanded to program the entire chip. The program is linked to run from the internal SRAM.

```
1   /*********************************************************************
2    * Bit definitions of status register
3    *********************************************************************/
4   #define FS_DONE                          0x00000001
5   #define FS_PROGGNT                       0x00000002
6   #define FS_RDY                           0x00000004
7   #define FS_ERR                           0x00000020
8
9
10  /*********************************************************************
11   * Bit definitions of control register
12   *********************************************************************/
13  #define  FS_CS                           0x00000001
14  #define  FS_WRE                          0x00000002
15  #define  FS_WEB                          0x00000004
16  #define  FS_RD_LATCH                     0x00000020
17  #define  FS_WPB                          0x00000080
18  #define  FS_SET_DATA                     0x00000400
19  #define  FS_RSSL                         0x00000800
20  #define  FS_PROG_REQ                     0x00001000
21  #define  FS_CLR_BUF                      0x00004000
22  #define  FS_LOAD_REQ                     0x00008000
23
24  /*********************************************************************
25   * Other
26   *********************************************************************/
27  #define FLASH_MEMORY_BASE                     0x10400000
28  #define FLASH_PAGE_SIZE_BYTES            512
29  #define FLASH_IMAGE_BASE                      0x10400000
30  #define RAM_BUFFER_BASE                            0x404000
31
32  #define FLASH_PROGRAM_CYCLES             72
33  #define FLASH_ERASE_CYCLES               9375
34  #define BUS_DECIMATOR                    60
35  #define WAIT_STATES                           0x8003
36
37  #define EN_T                                  0x00008000
38
39  /* Header file for LPC210X */
```

```
#include"LPC288x.h"


/*------------------- MAIN -------------------- */
void C_entry()
     {
          volatile unsigned char * FlashMemoryPointer;
          volatile unsigned int * LatchPointer;
          volatile unsigned int * Ramptr;
          unsigned int * Patch;
          unsigned int sec_size=0;
          int counter;


          /****** Flash Clock setup ******/
          /* The Flash module needs a 66KHz clock so the value set in the F_CLK_TIME register is 60
          */
          F_CTRL = (FS_WEB | FS_CS);
           while ( F_CLK_TIME != BUS_DECIMATOR)
           {
                F_CLK_TIME = BUS_DECIMATOR;
           }


          /****** Unprotecting Sector 0 ******/
         FlashMemoryPointer = (unsigned char *) FLASH_IMAGE_BASE;
         /* Writing to an address within the sector*/
          Patch = (unsigned int *) FlashMemoryPointer;
        *Patch = 0;
         /* Trigger value for Unprotect*/
        F_CTRL = (FS_LOAD_REQ | FS_WPB | FS_WEB | FS_WRE | FS_CS);


          /****** Erasing Sector 0 ******/
         /* Wait for Flash to be ready */
           while (( F_STAT & FS_RDY) != FS_RDY) { }
         /* Program Timer for erase */
        F_PROG_TIME= (FLASH_ERASE_CYCLES | EN_T);
         /* Trigger value for Erase */
        F_CTRL = (FS_PROG_REQ | FS_WPB | FS_CS);
         /* Wait for erase to complete */
        while (( F_STAT & FS_DONE) != FS_DONE) { }

          /****** Preparing Page buffer in SRAM ******/
         /* Setting ptr to base of RAM buffer */
          Ramptr= (unsigned int *) RAM_BUFFER_BASE;
         /* Loading 512 bytes */
      for (counter=0; counter<(FLASH_PAGE_SIZE_BYTES>>2); counter++)
           {
                *Ramptr = 0xCCCCBBBB;                                                    Ramptr++;
           }
```

```
91              /* Setting pointer for latches */
92              LatchPointer = (unsigned int *) FLASH_IMAGE_BASE;
93
94            /****** Sector 0 ********/
95            while(sec_size<=0xFFFF)
96            {
97             /* Wait for Flash to be ready */
98               while (( F_STAT & FS_RDY) != FS_RDY) { }
99
100              /* Setting pointer for RAM buffer */
101             Ramptr= (unsigned int *) RAM_BUFFER_BASE;
102              /****** Presetting Data latches ******/
103            /* Setting and Clearing FS_SET_DATA */
104              F_CTRL = (FS_WRE | FS_WEB | FS_CS | FS_SET_DATA); F_CTRL = (FS_WRE | FS_WEB | FS_CS);
105
106            /* Loading latches*/
107
108              for (counter=0; counter<(FLASH_PAGE_SIZE_BYTES>>2); counter++)
109                   {
110                        *LatchPointer = *Ramptr; LatchPointer++;
111                   sec_size=sec_size+4;
112                   Ramptr++;
113                   }
114
115              /****** Programming Page ********/
116            /* Program Timer for programming*/
117              F_PROG_TIME = (FLASH_PROGRAM_CYCLES | EN_T);
118            /* Trigger value for programming*/
119              F_CTRL = (FS_PROG_REQ | FS_WPB | FS_WRE | FS_CS);
120            /* Wait for programming to complete */
121              while (( F_STAT & FS_DONE) != FS_DONE) { }
122            /* Disable Timer*/
123             F_PROG_TIME = 0;                                          }
124
125           /* Do-nothing loop */
126              while(1){}
127
128         }
```

7. Legal information

7.1 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.

7.2 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 for 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.

7.3 Trademarks

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

8. Contents

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

Date of release: 8 January 2007
Document identifier: AN10548_1