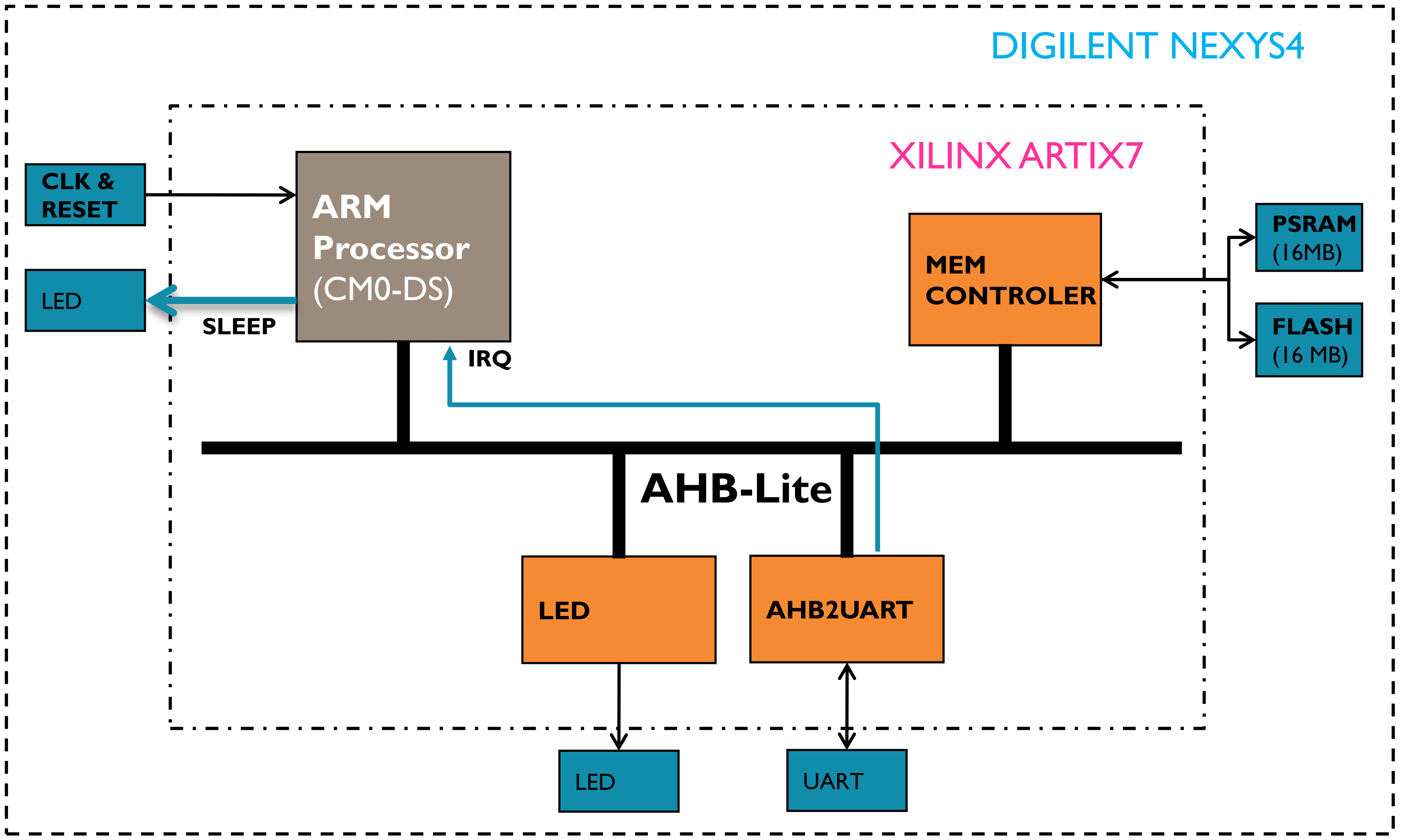
Programming in C and Interrupts

Design Summary

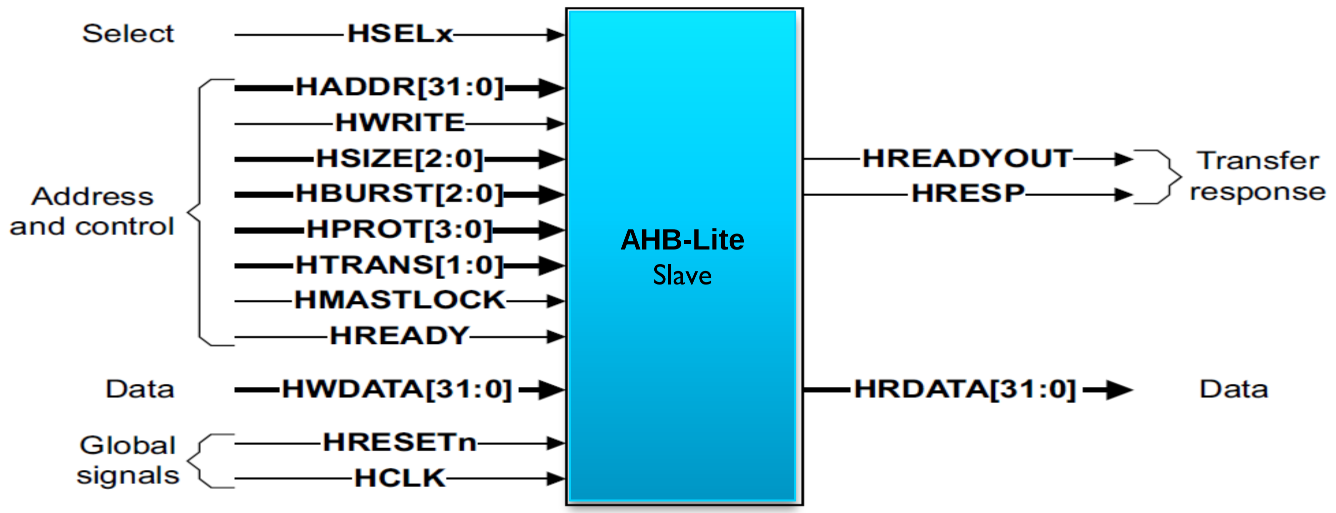


Design Summary

- The design consists of 3 peripherals with below memory map

	Base Address	Size
PSRAM/Internal Ram	0x0000_0000	8 MB/16KB
LED	0x5000_0000	0x00 General Purpose IO
UART	0x5100_0000	0x00 Send/Receive 0x04 Control Register Can generate Interrupt

AHB Slaves



```
AHBUART uAHBUART(
        .HCLK(HCLK),
        .HRESETn(HRESETn),
        .HADDR(HADDR[31:0]),
        .HTRANS(HTRANS[1:0]),
        .HWDATA(HWDATA[31:0]),
        .HWRITE(HWRITE),
        .HREADY(HREADY),
        .HREADYOUT(HREADYOUT_UART),
        .HRDATA(HRDATA_UART[31:0]),
        .HSEL(HSEL_UART),

        .RsRx(RsRx),
        .RsTx(RsTx),
        .uart_irq(UART_IRQ)
        );
```

```
assign          IRQ = {14'b0000_0000_0000_00,UART_IRQ,1'b0};
```

ARMv6-M Exception Model

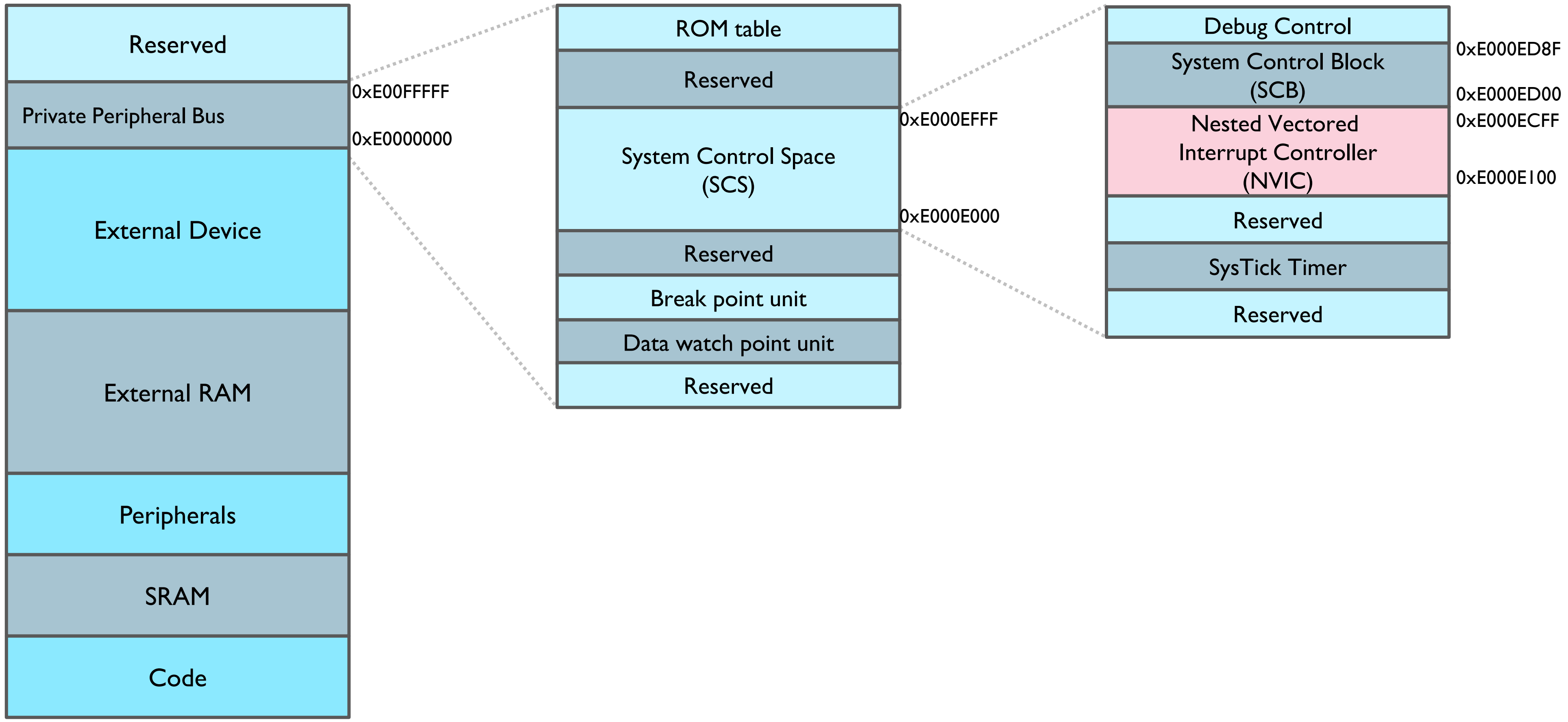
Exception number	Exception type	Priority
1	Reset	-3 (highest)
2	NMI	-2
3	HardFault	-1
4-10	Reserved	
11	SVCall	Programmable
12-13	Reserved	
14	PendSV	Programmable
15	SysTick, optional	Programmable
16 + N	External interrupt 0-31	Programmable

Cortex-M0 Interrupt Controller

- Nested Vectored Interrupt Controller (NVIC)
  - An integrated part of the Cortex-M0 processor;
  - Supports up to 32 IRQ inputs and a non-maskable interrupt (NMI) inputs;
  - Flexible interrupt management
  - Enable/ disable interrupt;
  - Pending control;
  - Priority configuration;
  - Hardware nested interrupt support;
  - Vectored exception entry;
  - Interrupt masking;
  - Can be easily accessed using C or assembly language.
  - Location: Private Peripheral Bus → System Control Space →NVIC

Cortex-M0 Interrupt Controller

- Memory map of Nested Vectored Interrupt Controller (NVIC)

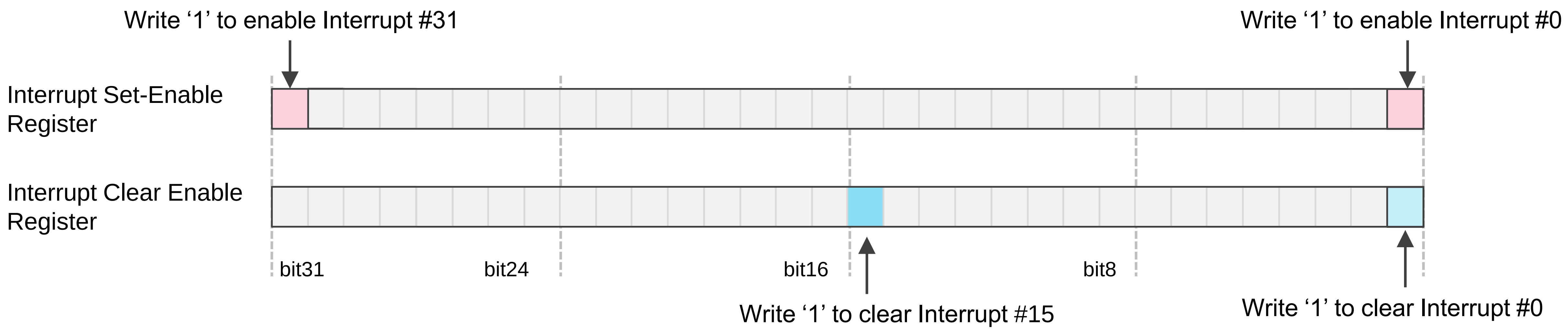


NVIC Registers

Address	Register
0xE000E100	**Interrupt Set-Enable Register**
0xE000E104 — 0xE000E17F	Reserved
0xE000E180	**Interrupt Clear-Enable Register**
0xE000E184 — 0xE000E1FF	Reserved
0xE000E200	**Interrupt Set-Pending Register**
0xE000E204 — 0xE000E27F	Reserved
0xE000E280	**Interrupt Clear-Pending Register**
0xE000E300 — 0xE000E3FC	Reserved
0xE000E400 — 0xE000E41C	**Interrupt Priority Registers**
0xE000E420 — 0xE000E43C	Reserved

NVIC Registers

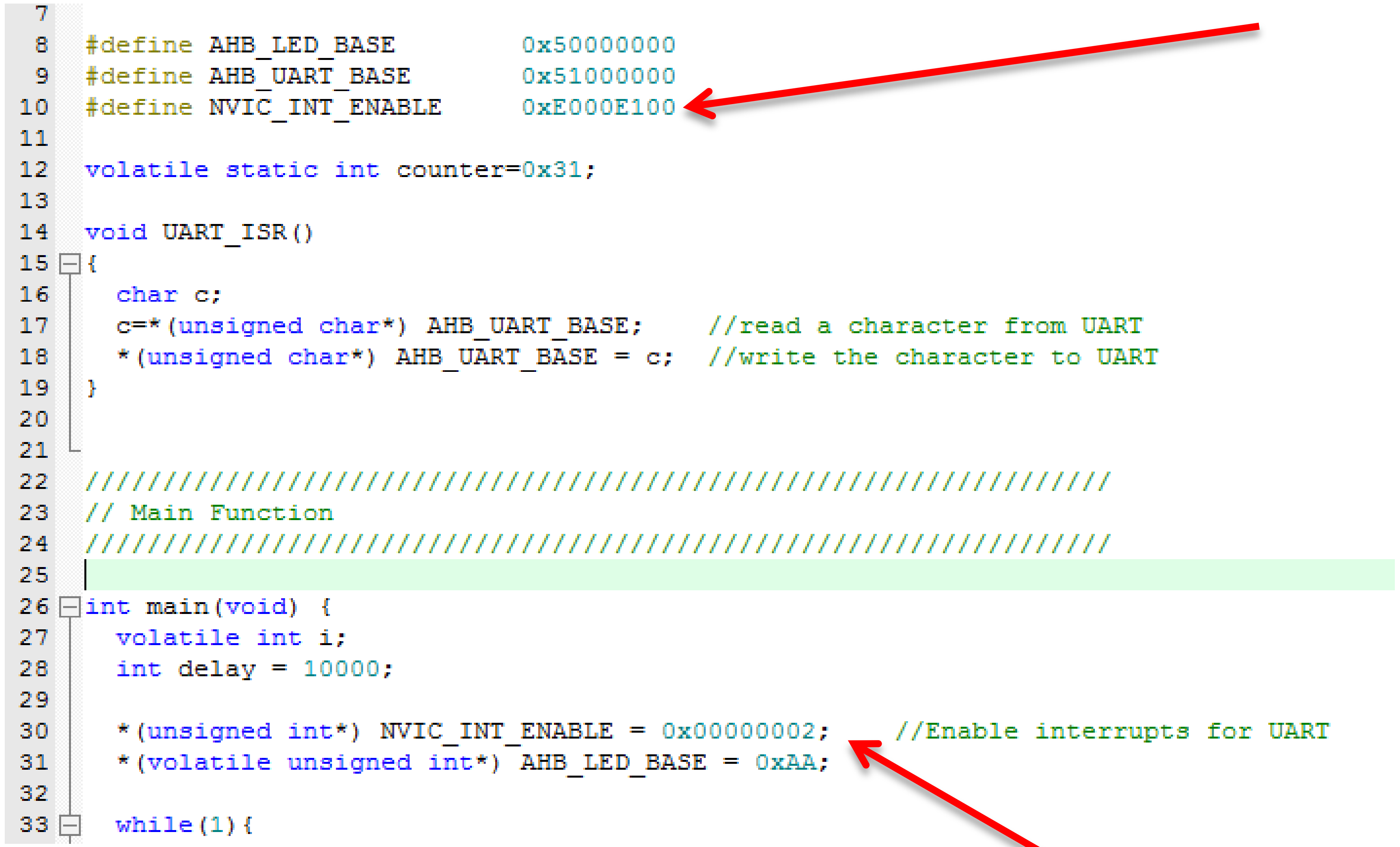
- Interrupt Set-Enable Register
  - Write ‘1’ to enable one or more interrupts;
  - Write ‘0’ has no effect;
- Interrupt Clear Enable Register
  - Write ‘1’ to Clear one or more interrupts.
  - Write ‘0’ has no effect;


Write ‘1’ to enable Interrupt #31
Write ‘1’ to enable Interrupt #0
Interrupt Set-Enable Register
Interrupt Clear Enable Register
bit31
bit24
bit16
bit8
Write ‘1’ to clear Interrupt #15
Write ‘1’ to clear Interrupt #0

NVIC Registers

- Why use separated register address
  - Compared with the "read-modify-write" process the benefit of using separated address includes:
  - Reduces the steps needed for enabling/ disabling an interrupt, resulting in smaller code and less execution time;
  - Prevents the race condition, e.g. the main thread is accessing a register by "read-modify-write" process, and it is interrupted between its "read" and "write" operation. If the ISR again modifies the same register that is currently being accessed by the main thread, a conflict will occur.
- Interrupt pending and clear pending
  - An interrupt goes into pending status if it cannot be processed immediately, e.g. a lower priority interrupt will be pended if a higher interrupt is being processed.

Enabling UART Interrupt



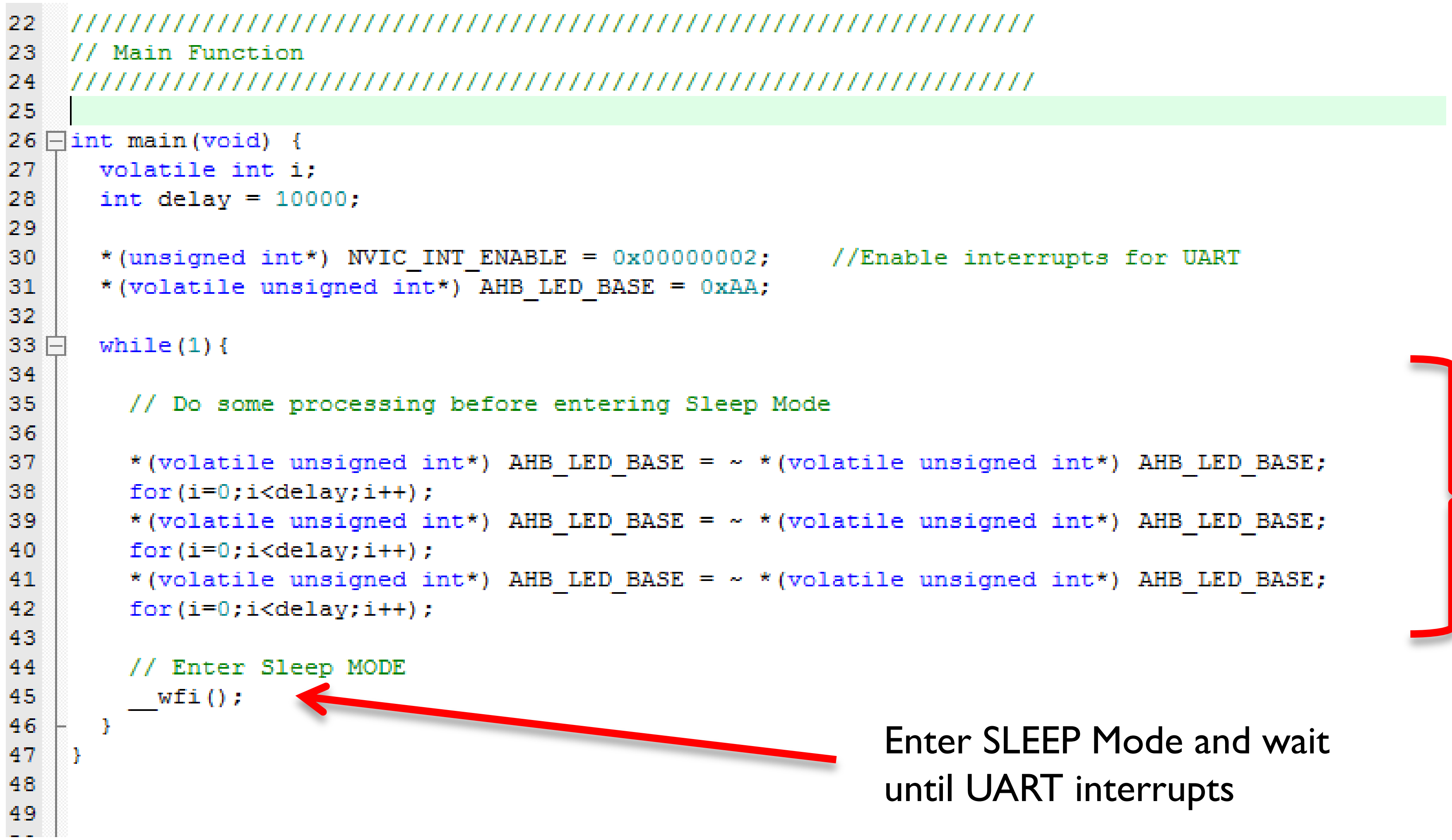
```
#define AHB_LED_BASE        0x50000000
#define AHB_UART_BASE       0x51000000
#define NVIC_INT_ENABLE     0xE000E100

volatile static int counter=0x31;

void UART_ISR()
{
  char c;
  c=*(unsigned char*) AHB_UART_BASE;    //read a character from UART
  *(unsigned char*) AHB_UART_BASE = c;  //write the character to UART
}


//////////////////////////////////////////////////////////////////
// Main Function
//////////////////////////////////////////////////////////////////

int main(void) {
  volatile int i;
  int delay = 10000;

  *(unsigned int*) NVIC_INT_ENABLE = 0x00000002;     //Enable interrupts for UART
  *(volatile unsigned int*) AHB_LED_BASE = 0xAA;

  while(1){
```

WFI (Wait for Interrupt) Instruction

- When the processor executes a WFI it stops executing instructions and enters sleep mode.
- The Processor stays in SLEEP mode until one of the following event occurs,
  - a non-masked interrupt occurs and is taken
  - an interrupt masked by PRIMASK becomes pending
  - a Debug Entry request.

- When the processor enters "SLEEP" mode, the "SLEEPING" sideband signal is asserted from Cortex M0
- This signal is used to implement various hardware power saving techniques

WFI (Wait for Interrupt) Instruction



```
//////////////////////////////////////////////////////////////////////
// Main Function
//////////////////////////////////////////////////////////////////////

int main(void) {
  volatile int i;
  int delay = 10000;

  *(unsigned int*) NVIC_INT_ENABLE = 0x00000002;    //Enable interrupts for UART
  *(volatile unsigned int*) AHB_LED_BASE = 0xAA;

  while(1){

    // Do some processing before entering Sleep Mode

    *(volatile unsigned int*) AHB_LED_BASE = ~ *(volatile unsigned int*) AHB_LED_BASE;
    for(i=0;i<delay;i++);
    *(volatile unsigned int*) AHB_LED_BASE = ~ *(volatile unsigned int*) AHB_LED_BASE;
    for(i=0;i<delay;i++);
    *(volatile unsigned int*) AHB_LED_BASE = ~ *(volatile unsigned int*) AHB_LED_BASE;
    for(i=0;i<delay;i++);

    // Enter Sleep MODE
    __wfi();
  }
}
```

Processor Busy

Enter SLEEP Mode and wait
until UART interrupts

Calling a C Function from Assembly

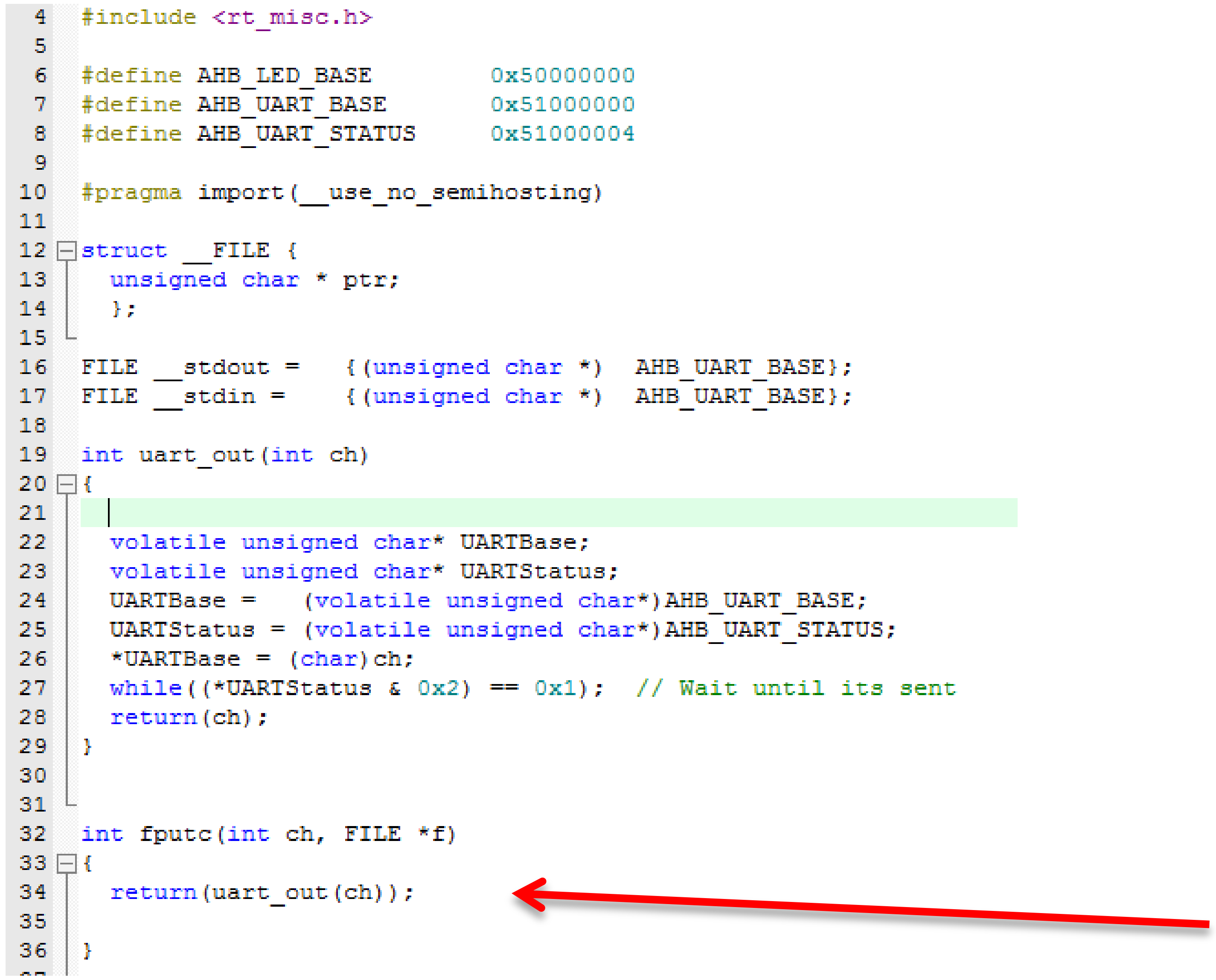
- ISR can be written in either assembly or C language, for example in C:

```
void UART_ISR() {
     char c;
     c=*(char*) AHB_UART_BASE;     //read a character from UART
     …
}
```

- Call a C function from the assembly code, for example:

```
UART_Handler      PROC
                  EXPORT UART_Handler   // label name in assembly
                  IMPORT UART_ISR       // function name in C
                  PUSH    {R0,R1,R2,LR} // context saving
                  BL UART_ISR       // branch to ISR written in C
                  POP     {R0,R1,R2,PC} // context restoring
                  ENDP
```

Retarget printf



```
#include <rt_misc.h>

#define AHB_LED_BASE        0x50000000
#define AHB_UART_BASE       0x51000000
#define AHB_UART_STATUS     0x51000004

#pragma import(__use_no_semihosting)

struct __FILE {
  unsigned char * ptr;
  };

FILE __stdout =    {(unsigned char *)  AHB_UART_BASE};
FILE __stdin =     {(unsigned char *)  AHB_UART_BASE};

int uart_out(int ch)
{

  volatile unsigned char* UARTBase;
  volatile unsigned char* UARTStatus;
  UARTBase =    (volatile unsigned char*)AHB_UART_BASE;
  UARTStatus = (volatile unsigned char*)AHB_UART_STATUS;
  *UARTBase = (char)ch;
  while((*UARTStatus & 0x2) == 0x1);  // Wait until its sent
  return(ch);
}


int fputc(int ch, FILE *f)
{
  return(uart_out(ch));

}
```

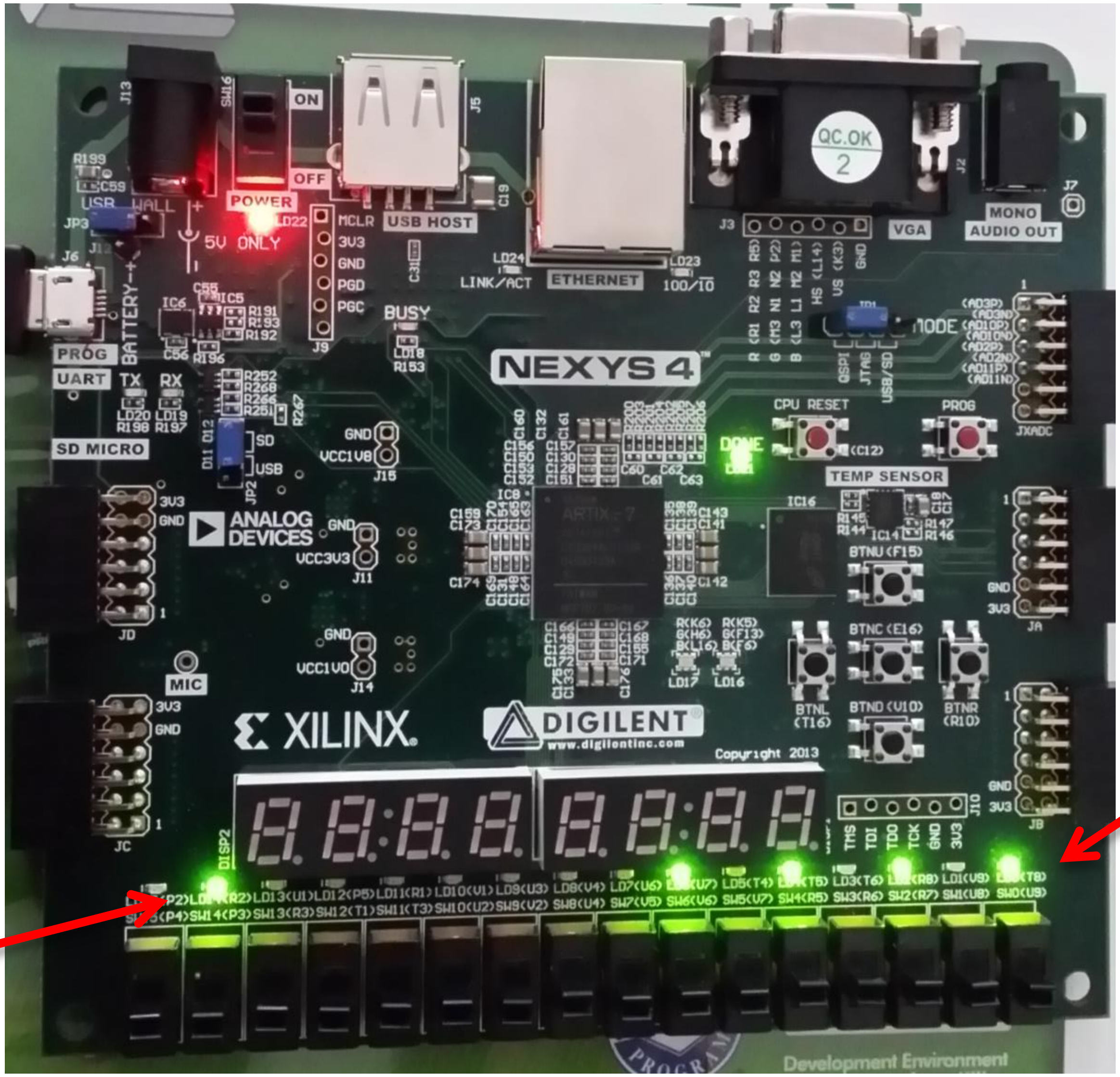
Lab Steps (with PSRAM)

1. Compile the Software using KEIL MDK ARM and generate code.hex file
2. Follow the steps given in the lab manual to download code.hex onto PSRAM
3. Open FPGA project under Vivado and implement the design
4. Use Vivado hardware manager to download the .bit file
5. Communicate with the board using HyperTerminal (or any other serial terminal)

Lab Steps (with BRAM)

1. Compile the Software using KEIL MDK ARM and generate code.hex file
2. Open FPGA project under Vivado and implement the design
3. Use Vivado hardware manager to download the .bit file
4. Communicate with the board using HyperTerminal (or any other serial terminal)

Output



LEDs change pattern when you send UART Characters.

SLEEPING SIGNAL