VISVESVARAYA TECHNOLOGICAL UNIVERSITY
BELGAUM

MICROCONTROLLER AND EMBEDDED SYSTEMS LABORATORY (18CSL48)

(As per Visvesvaraya Technological University Syllabus)

Complied By:

Prof. Karthik D U
Assistant Professor, Dept. of CSE

Prof. PrashanthKumar S P
Assistant Professor, Dept. of CSE

Prof. Shilpashree S
Assistant Professor, Dept. of CSE

Prof. SathiyaRaj
Assistant Professor, Dept. of CSE

Mr. Shivaprasad
Lab Instructor, Dept. of CSE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
ACHARYA INSTITUTE OF TECHNOLOGY
(Affiliated to VTU, Belgaum, Approved by AICTE, New Delhi and Govt. of Karnataka),
Acharya Dr. Sarvepalli Radhakrishnan Road, Bangalore-560107.
Ph: 91-080-28396011, 23723466, 28376431
URL: www.acharya.ac.in

2019-20

MICROCONTROLLER AND EMBEDDED SYSTEMS LABORATORY

(Effective from the academic year 2019 -2020)

Course Code	18CSL48	**CIE Marks**	40
Number of Contact Hours/Week	0:2:2	**SEE Marks**	60
Total Number of Lab Contact	36	**Exam Hours**	03

CREDITS–02

Course Learning Objectives: This course (18CSL48) will enable students to:

- Develop and test Program using ARM7TDMI/LPC2148
- Conduct the experiments on an ARM7TDMI/LPC2148 evaluation board using evaluation version of Embedded 'C' &Keil Uvision-4 tool/compiler.

Programs List:

PART A Conduct the following experiments by writing program using ARM7TDMI/LPC2148 using an evaluation board/simulator and the required software tool.

1.	Write a program to multiply two 16 bit binary numbers.
2.	Write a program to find the sum of first 10 integer numbers.
3.	Write a program to find factorial of a number.
4.	Write a program to add an array of 16 bit numbers and store the 32 bit result in internal RAM
5.	Write a program to find the square of a number (1 to 10) using look-up table.
6.	Write a program to find the largest/smallest number in an array of 32 numbers.
7.	Write a program to arrange a series of 32 bit numbers in ascending/descending order.
8.	Write a program to count the number of ones and zeros in two consecutive memory locations.

PART –B Conduct the following experiments on an ARM7TDMI/LPC2148 evaluation board using evaluation version of Embedded 'C' & Keil Uvision-4 tool/compiler.

9.	Display “Hello World” message using Internal UART.
10.	Interface and Control a DC Motor.
11.	Interface a Stepper motor and rotate it in clockwise and anti-clockwise direction.
12.	Determine Digital output for a given Analog input using Internal ADC of ARM controller.
13.	Interface a DAC and generate Triangular and Square waveforms.
14.	Interface a 4x4 keyboard and display the key code on an LCD.
15.	Demonstrate the use of an external interrupt to toggle an LED On/Off.
16.	Display the Hex digits 0 to F on a 7-segment LED interface, with an appropriate delay in between

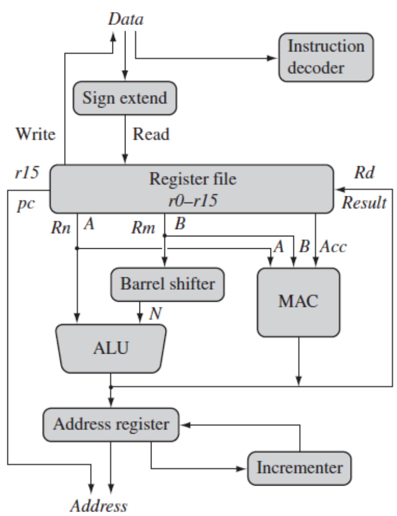
Laboratory Outcomes: The student should be able to:
• Develop and test program using ARM7TDMI/LPC2148 • Conduct the following experiments on an ARM7TDMI/LPC2148 evaluation board using evaluation version of Embedded 'C' &Keil Uvision-4 tool/compiler.
Conduct of Practical Examination:
• Experiment distribution o For laboratories having only one part: Students are allowed to pick one experiment from the lot with equal opportunity. o For laboratories having PART A and PART B: Students are allowed to pick one experiment from PART A and one experiment from PART B, with equal opportunity. • Change of experiment is allowed only once and marks allotted for procedure to be made zero of the changed part only. • Marks Distribution *(Coursed to change in accordance with university regulations)* g)For laboratories having only one part – Procedure + Execution + Viva-Voce: 15+70+15 = 100 Marks h)For laboratories having PART A and PART B i.Part A – Procedure + Execution + Viva = 6 + 28 + 6 = 40 Marks ii.Part B – Procedure + Execution + Viva = 9 + 42 + 9 = 60 Marks

ARM PROCESSOR FUNDAMENTALS

- ✓ ARM core has functional units connected by data buses, as shown in Figure where, the arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area. Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item.
  The instruction decoder translates instructions before they are executed.

- ✓ Data items are placed in the *register file*—a storage bank made up of 32-bit registers.ARM instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*. Source operands are read from the register file using the internal buses *A* and *B*, respectively.
  The ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values*Rn* and *Rm* from the *A* and *B* buses and computes a result. Data processing instructions write the result in *Rd* directly to the register file. Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the *Address* bus.

- ✓ One important feature of the ARM is that register *Rm* alternatively can be pre-processed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.
  After passing through the functional units, the result in *Rd* is written back to the register file using the *Result* bus. For load and store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location



ARM core dataflow model.

Registers

General-purpose registers hold either data or an address. They are identified with the letter *r* prefixed to the register number. For example, register 4 is given the label *r4.* Figure shows the active registers available in *user* mode—a protected mode normally used when executing applications. The processor can operate in seven different modes. All the registers shown are 32 bits in size.

There are up to 18 active registers: 16 data registers and 2 processor status registers. The data registers are visible to the programmer as *r0* to *r15.*

The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15.*

- Register *r13* is traditionally used as the stack pointer (*sp*) and stores the head of the stack

in the current processor mode.

- Register *r14* is called the link register (*lr*) and is where the core puts the return address

whenever it calls a subroutine.

- Register *r15* is the program counter (*pc*) and contains the address of the next instruction

to be fetched by the processor.

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 sp
r14 lr
r15 pc

cpsr
-

Registers available in *user* mode.

In ARM state the registers *r0* to *r13* are *orthogonal*—any instruction that you can apply to *r0* you can equally well apply to any of the other registers. In addition to the 16 data registers, there are two program status registers: *cpsr*and *spsr*(the current and saved program status registers, respectively)

Current Program Status Register

- The ARM core uses the *cpsr*to monitor and control internal operations.
- The *cpsr*is a dedicated 32-bit register and resides in the register file. Figure shows the basic layout of a generic program status register. The shaded parts are reserved for future expansion.
- The *cpsr*is divided into four fields, each 8 bits wide: flags, status, extension, and control.
- In current designs the extension and status fields are reserved for future use.
- The control field contains the processor mode, state, and interrupt mask bits.
- The flags field contains the condition flags.

Some ARM processor cores have extra bits allocated. For example, the *J* bit, which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8-bit instructions..

Processor Modes

The processor mode determines which registers are active and the access rights to the *cpsr*register itself. Each processor mode is either privileged or nonprivileged: A privileged mode allows full read-write access to the *cpsr*. Conversely, a nonprivileged mode only allows read access to the control field in the *cpsr*but still allows read-write access to the condition flags.

There are seven processor modes in total: six privileged modes (*abort*, *fast interrupt, request*, *interrupt request*, *supervisor*, *system*, and *undefined*) and one nonprivileged mode (*user*).

- The processor enters *abort* mode when there is a failed attempt to access memory
- *Fast interrupt request* and *interrupt request* modes correspond to the two interrupt levels available on the ARM processor.
- *Supervisor* mode is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
- *System* mode is a special version of *user* mode that allows full read-write access to the *cpsr*.
- *Undefined* mode is used when the processor encounters an instruction that is undefined or not supported by the implementation.
- *User* mode is used for programs and applications.

Banked Registers

Figure shows all 37 registers in the register file. Of those, 20 registers are hidden from a program at different times. These registers are called *banked registers* and are identified by the shading in the diagram. They are available only when the processor is in a particular mode; for example, *abort* mode has banked registers *r13_abt*, *r14_abt* and *spsr_abt.*

Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or *_mode*. Every processor mode except *user* mode can change mode by writing directly to the mode bits of the *cpsr.*

All processor modes except *system* mode have a set of associated banked registers that are a subset of the main 16 registers. A banked register maps one-tone onto a *user* mode register.

If you change processor mode, a banked register from the new mode will replace an existing register. For example, when the processor is in the *interrupt request* mode, the instructions you execute still access registers named *r13* and *r14*. However, these registers are the banked registers *r13_irq* and *r14_irq*. The *user* mode registers *r13_usr* and *r14_usr* are not affected by the instruction referencing these registers. A program still has normal access to the other registers *r0* to *r12*.

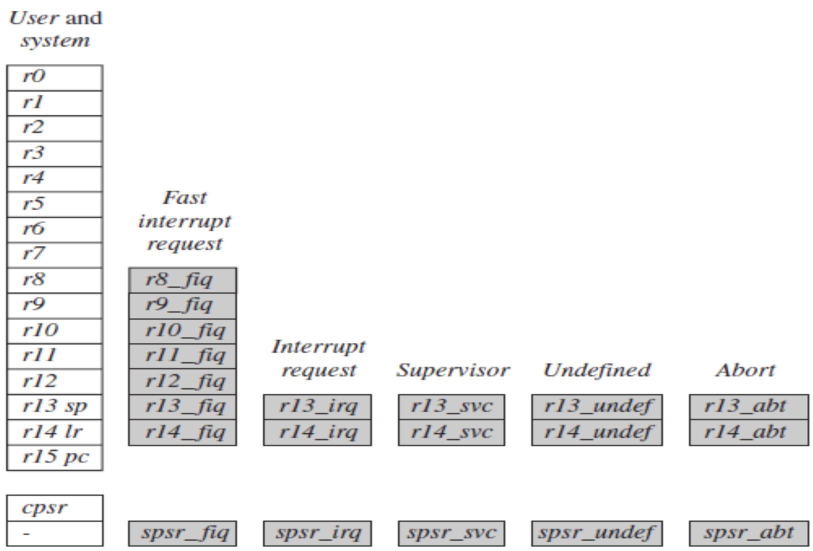
The processor mode can be changed by a program that writes directly to the *cpsr*(the processor core has to be in privileged mode) or by hardware when the core responds to an exception or interrupt. The following exceptions and interrupts cause a mode change:
reset, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*,and *undefined instruction*

Figure illustrates what happens when an interrupt forces a mode change. The figure shows the core changing from *user* mode to *interrupt request* mode, which happens when an *interrupt request* occurs due to an external device raising an interrupt to the processor core.
This change causes*user* registers *r13* and *r14* to be banked. The *user* registers are replaced with registers *r13_irq* and *r14_irq*, respectively. Note *r14_irq* contains the return address and *r13_irq* contains the stack pointer for *interrupt request* mode.

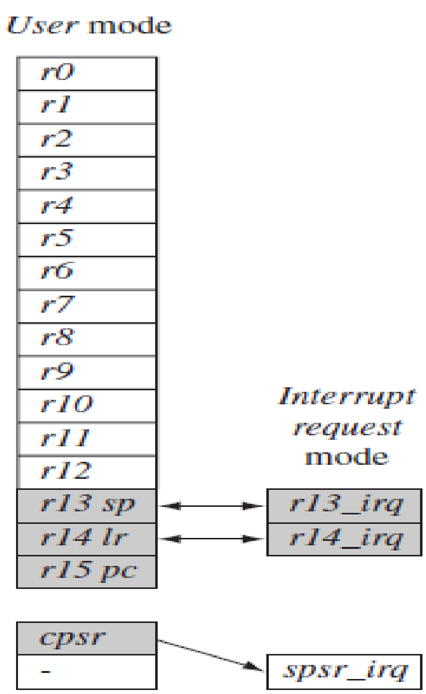
A new register appears in *interrupt request* mode:

- The saved program status register *(spsr)*, stores the previous mode's *cpsr.*
- The *cpsr is* copied into *spsr_irq*
- To return back to *user* mode, a special return instruction is used that instructs the core to restore the original *cpsr*from the *spsr_irq*and bank in the *user* registers *r13* and *r14*.

- The *spsr*can only be modified and read in a privileged mode. There is no *spsr*available in *user* mode.
- The *cpsr*is not copied into the *spsr*when a mode change is forced due to a program writing directly to the *cpsr*.
- The saving of the *cpsr*only occurs when an exception or interrupt is raised.
- The current active processor mode occupies the five least significant bits of the *cpsr*.
- When power is applied to the core, it starts in *supervisor* mode, which is privileged.
- Starting in a privileged mode is useful since initialization code can use full access to the *cpsr*to set up the stacks for each of the other modes.



Complete ARM register set.



Changing mode on an exception

Table lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the *cpsr.*

Processor mode.

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

State and Instruction Sets

The state of the core determines which instruction set is being executed. There are three instruction sets:
ARM, Thumb and Jazelle.

- The ARM instruction set is only active when the processor is in ARM state.
- The Thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions. You cannot intermingle sequential ARM, Thumb, and Jazelle Instructions.
- The Jazelle*J* and Thumb *T* bits in the *cpsr*reflect the state of the processor.

When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions. This is the case when power is applied to the processor. When the T bit is 1, then the processor is in Thumb state. To change states the core executes a specialized branch instruction

Table compares the ARM and Thumb instruction set features.
The ARM designers introduced a third instruction set called *Jazelle. Jazelle*executes
8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java byte codes

ARM and Thumb instruction set features.

	ARM (*cpsr* $T = 0$)	Thumb (*cpsr* $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +*pc*	8 general-purpose registers +7 high registers +*pc*

Jazelle instruction set features.

	Jazelle (*cpsr* $T = 0, J = 1$)
Instruction size	8-bit
Core instructions	Over 60% of the Java bytecodes are implemented in hardware; the rest of the codes are implemented in software.

Interrupt Masks

Interrupt masks are used to stop specific interrupt requests from interrupting the processor.

There are two interrupt request levels available on the ARM processor core—*interrupt request* (IRQ) and *fast interrupt request* (FIQ).

The *cpsr*has two interrupt mask bits, 7 and 6 (or *I* and *F*), which control the masking of IRQ and FIQ, respectively. The *I*bit masks IRQ when set to binary 1, and similarly the *F* bit masks FIQ when set to binary 1.

Condition Flags

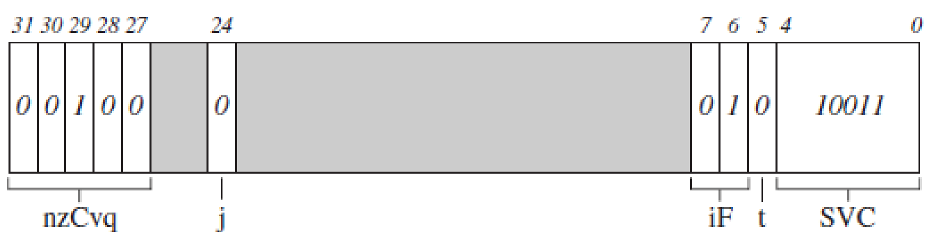
Condition flags are updated by comparisons and the result of ALU operations that specify the S instruction suffix. For example, if a SUBS subtract instruction results in a register value of zero, then the *Z* flag in the *cpsr*is set. This particular subtract instruction specifically updates the *cpsr.*

Flag	Flag name	Set when
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero, frequently used to indicate equality
N	Negative	bit 31 of the result is a binary 1

With processor cores that include the DSP extensions, the *Q* bit indicates if an overflow or saturation has occurred in an enhanced DSP instruction. The flag is "sticky" in the sense that the hardware only sets this flag. To clear the flag you need to write to the *cpsr*directly.

In Jazelle-enabled processors, the *J* bit reflects the state of the core; if it is set, the core is in Jazelle state. The *J* bit is not generally usable and is only available on some processor cores.To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems.

Figure shows a typical value for the *cpsr*with both DSP extensions and Jazelle.
In the *cpsr*example shown in Figure , the *C* flag is the only condition flag set. The rest *nzvq*flags are all clear. The processor is in ARM state because neither the Jazelle*j* or Thumb *t* bits are set. The IRQ interrupts are enabled, and FIQ interrupts are disabled. Finally the processor is in *supervisor* (*SVC*) mode since the mode[4:0] is equal to binary 10011.



Example: ***cpsr*= *nzCvqjiFt_SVC*.**

ARMinstructionset.

Mnemonics	**ARM ISA**	**Description**
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation

CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCR	R v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRR	C v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (cpsr or spsr)
MSR	v3	move to a status register (cpsr or spsr) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLAxy	v5E	signed multiply accumulate instructions ((16 × 16) + 32 = 32-bit)
SMLAL	v3M	signed multiply accumulate long ((32 × 32) + 64 = 64-bit)

SMLALxy	v5E	signed multiply accumulate long ((16 × 16) + 64 = 64-bit)
SMLAWy	v5E	signed multiply accumulate instruction (((32 × 16) 16) + 32 = 32-bit)
SMULL	v3M	signed multiply long (32 × 32 = 64-bit)
SMULxy SMULWy STC STC2	v5E v5E v2 v5	signed multiply instructions (16 × 16 = 32-bit) signed multiply instruction ((32 × 16) 16 = 32-bit) store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long ((32 × 32) + 64 = 64-bit)
UMULL	v3M	unsigned multiply long (32 × 32 = 64-bit)

INTRODUCTION TO KEIL SOFTWARE

The µVision4 IDE isa Windows- based software development platform that combines a robust editor, project manager, and makes facility. µVision4 integrates all tools including the C compiler, macro assembler, linker/locator, and HEX file generator. µVision4 helps expedite the development process of your embedded applications by providing the following:

- ✓ Full-featured source code editor,
- ✓ Device database for configuring the development tool setting,
- ✓ Project manager for creating and maintaining your projects,
- ✓ Integrated make facility for assembling, compiling, and linking your embedded applications,
- ✓ Dialogs for all development tool settings,
- ✓ True integrated source-level Debugger with high-speed CPU and peripheral simulator,
- ✓ Advanced GDI interface for software debugging in the target hardware and for

connection to Keil ULINK,

- ✓ Flash programming utility for downloading the application program into Flash ROM, Links to development tools manuals, device datasheets & user's guides.

The µVision4 IDE offers numerous features and advantages that help you quickly and successfully develop embedded applications. They are easy to use and are guaranteed to help you achieve your design goals.

Theinstallation steps for keil software are given below:

1. Double click on Keil µvision exe file.
2. click on **Next.**
3. Tick the check box towards to license agreements and click **Next.**
4. Select Destination folder and click **Next.**
5. To fill the names and e-mail ID in the text boxes then click **Next.**
6. Finally click on **Finish.**

The keil software flow

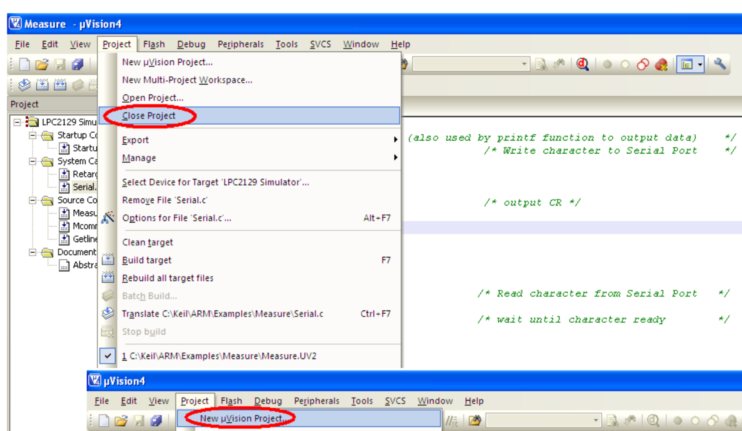
Click on the icon keil µVision4 and the follow the steps as given below.

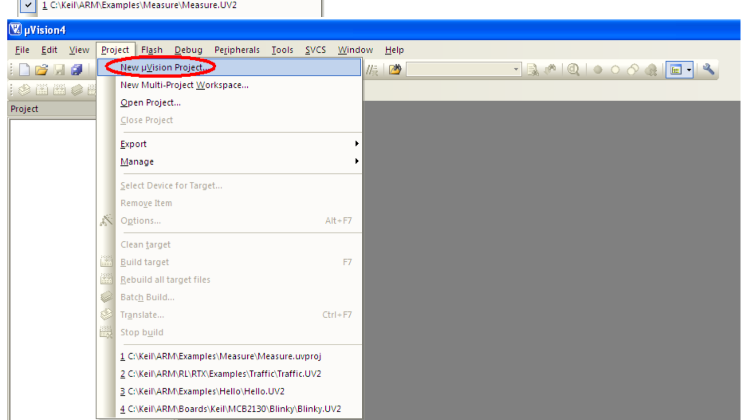
The menu bar provides with menus for editor operations, project maintenance, development tool option settings, program debugging, external tool control, window selection and manipulation, andon-linehelp. Thetool bar button sallow t o rapidly execute µVision4 commands. A Status Bar provides editor and debugger information. The various toolbars and thestatus bar can beenabled or disabled from the ViewMenu commands.

Creating a Project

Thefollowing are the stepsrequiredto execute program in keil software

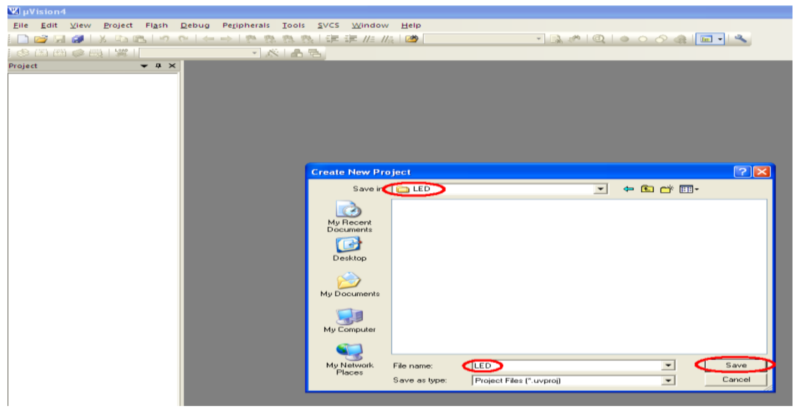
STEP 1: Go to "**Project**" and close the current project "**Close Project**".



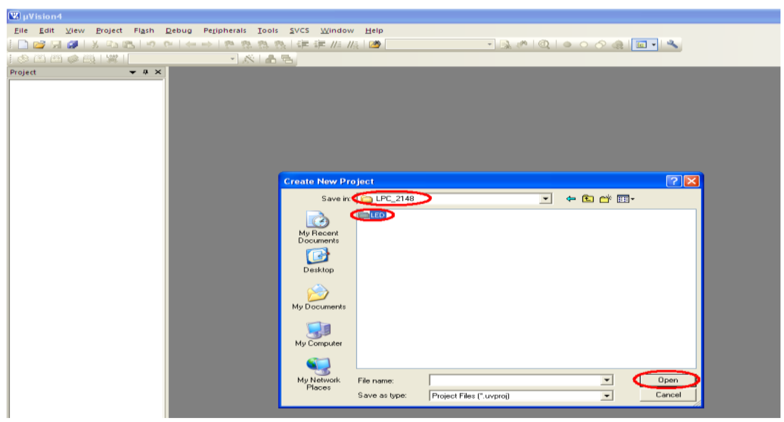


STEP 2: Go to the "**Project**" and click on "**New Micro vision Project**"

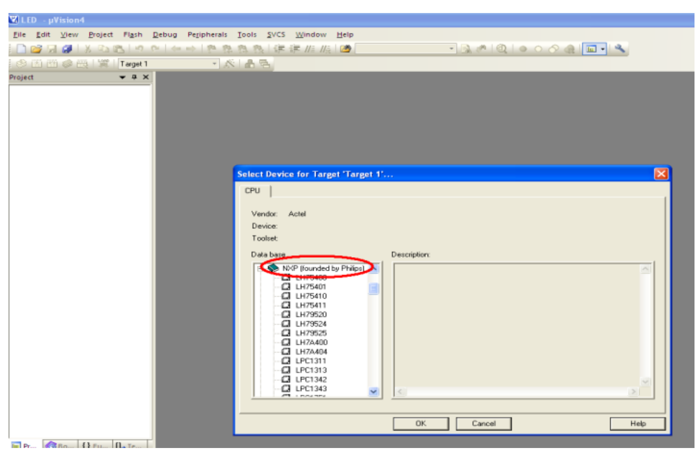
STEP3:Here we can see a small windows "**Create New Project**" and here we can select our destination to wherever we want.



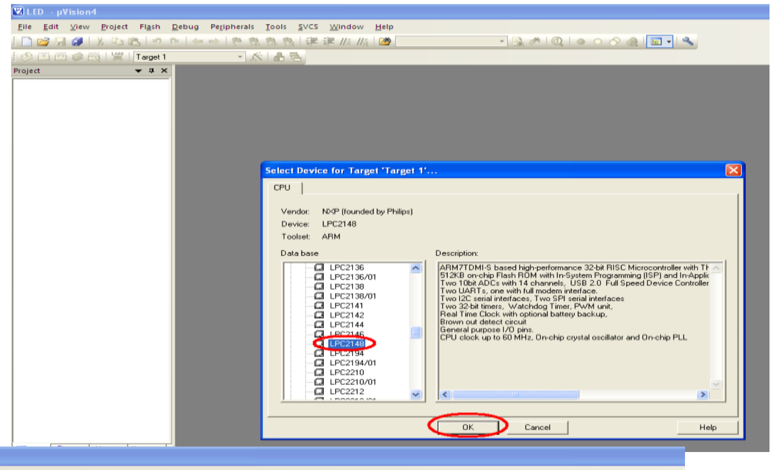
STEP4:Open one drive and create a"New Folder"and to give any name related to the Project.



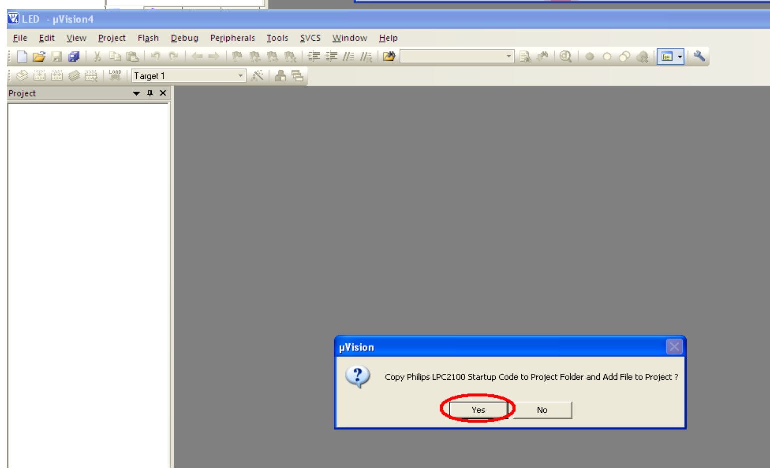
STEP5: Again we can see a small window as "**Select Device for Target 'Target 1'**, here select NXP founded by **Philips.**



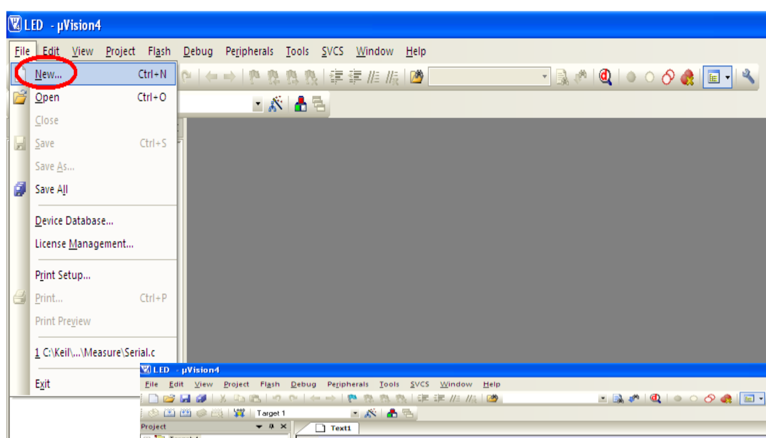
STEP 6: Within the NXP **Philips** select **LPC2148**



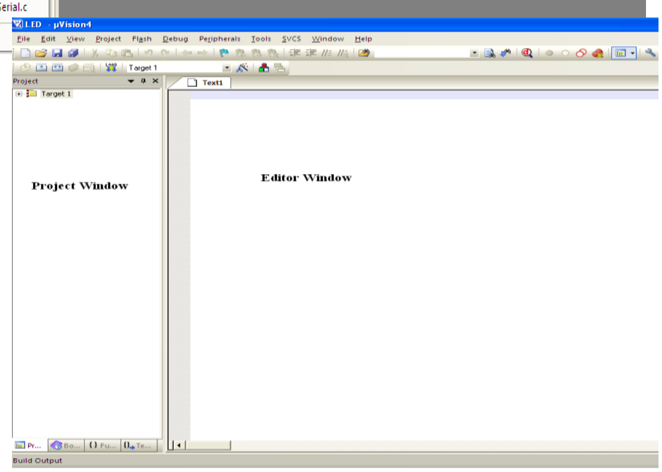
STEP7:For addingStartup file click on "Yes" or else click on "No".
(startup.s file is required only for C programs for assembly language program delete startup.s file)



STEP 8: Next go to "**File**" and click on "**New**".

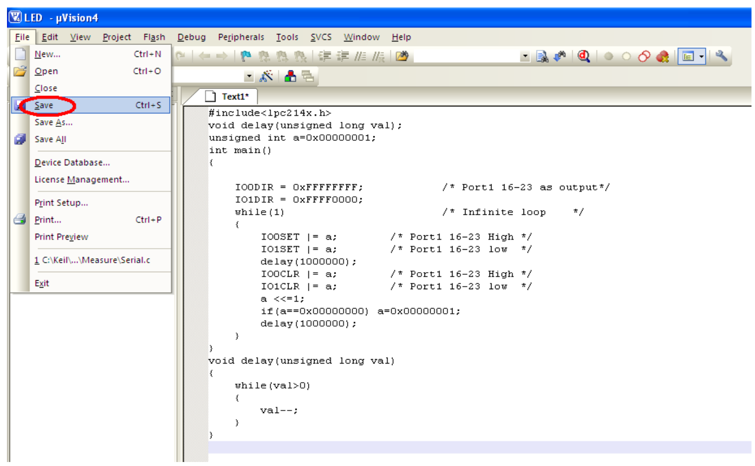


STEP9:These are the main three windows in the keil IDE. One is Project Workspace,

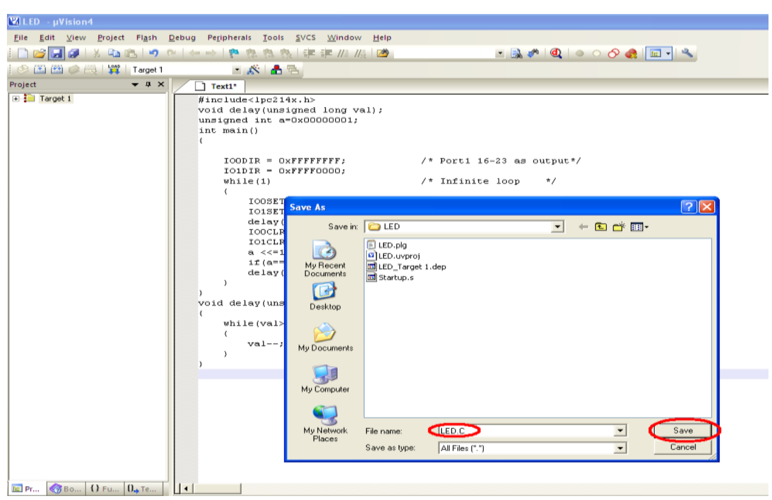


second is Editor Window and third is Output Window.

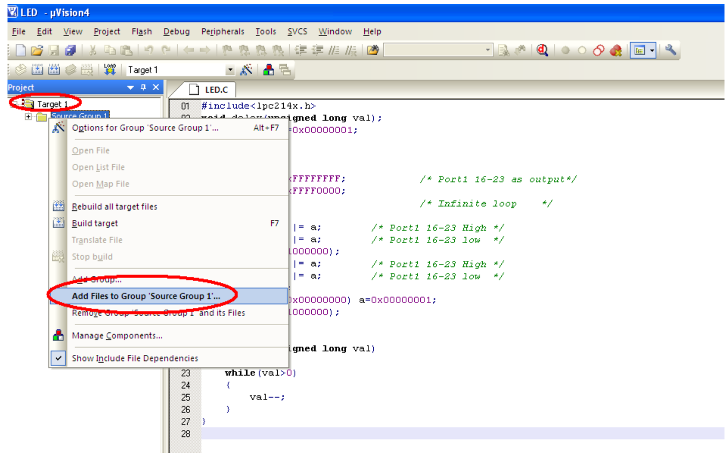
STEP10:In editor window start to write program and after editing we have to save.



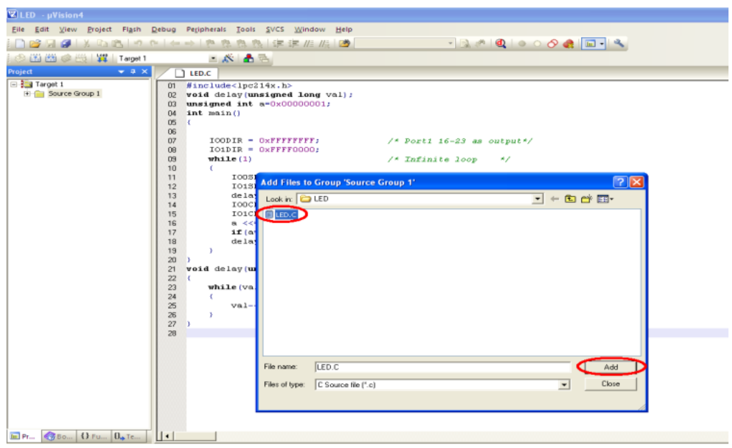
STEP11:Save the file, if the programisin “**C**” save as “**filename.c**” or else if it is an **assembly program** save as “**filename.s**”.



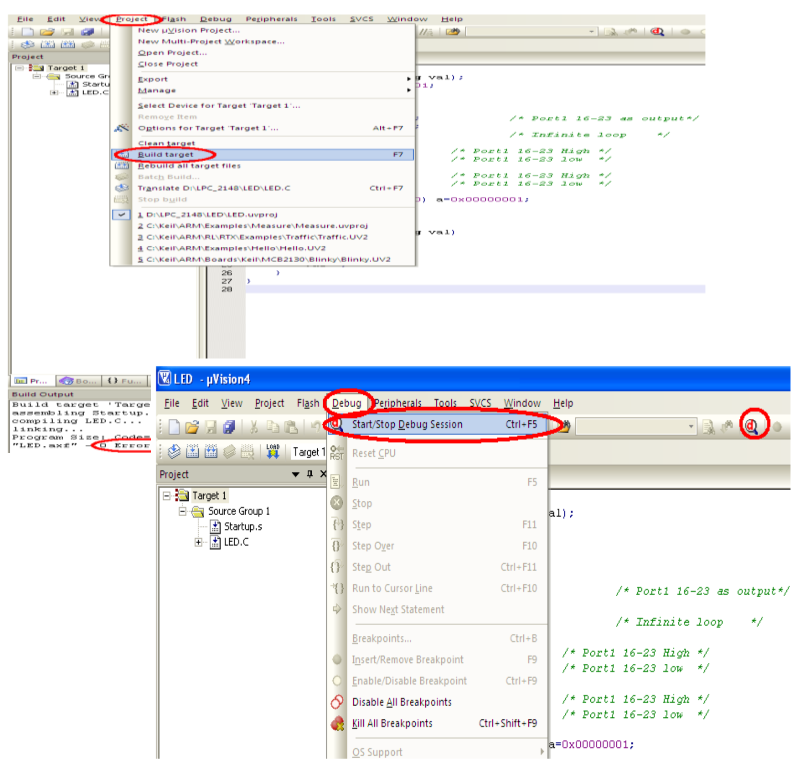
STEP 12: Next add this source file to Group1, for that go to "**Project Workspace**"drag the"**Target1**"in the right click on "**SourceGroup1**" and click on "**Add Files to Group "Source Group1"**.



STEP13: Here one small window will open as "**Add Files to Group" Source Group1", default** the Files of type will be in**C source Files (*.C).**If our program is in C we need to select **C source Files(*.C**) or select **ASM Source file (*.s,*.src,*.a*).**

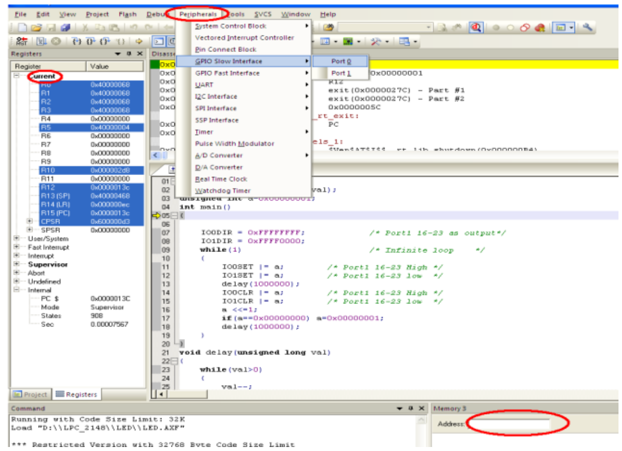


STEP14:Then go to"**Project**"click on"**Build Target**" or **F7**. There we can see errors and warning in Output Window.



SIMULATION PROGRAMS

STEP15:Goto “**Project**” click on “**Rebuild all target Files**” and start **Debug.** From **View** menu we can get **Memory Window.**



PART A - Software Programs

1. Write a program to multiply two 16 bit binary numbers.

```
        AREA MULTIPLY, CODE, READONLY

ENTRY                                   ; Mark first instruction to execute

START
        MOV r1,#0X1900                  ; STORE FIRST NUMBER IN R1
        MOV r2,#0X0C80                  ; STORE SECOND NUMBER IN R2
        MUL r3,r1,r2                    ; MULTIPLICATION

STOP B STOP
        END                             ; Mark end of file
```

Output:

1st Input : Register R1
2nd Input : Register R2
Result : Register R3

2. Write a program to find the sum of first 10 integer numbers.

```
        AREA SUM, CODE, READONLY

ENTRY                                   ; Mark first instruction to execute
        MOV R1, #10                     ; load 10 to register
        MOV R2, #0                      ; empty the register to store result

LOOP
        ADD R2, R2, R1          ; add the content of R1 with result at R2
        SUBS R1, #0x01                  ; Decrement R1 by 1
        BNE LOOP                        ; repeat till r1 goes 0

STOP B STOP
        END                             ; Mark end of file
```

Output:

Result can be viewed in RegisterR2 in hex decimal values.

3. Write a program to find factorial of a number.

```
        AREA  FACTORIAL, CODE, READONLY

ENTRY                                   ; Mark first instruction to execute

START

        MOV R0, #7                      ; STORE FACTORIAL NUMBER IN R0
        MOV R1, R0                      ; MOVE THE SAME NUMBER IN R1

FACT    SUBS R1, R1, #1                 ; SUBTRACTION
        CMP R1, #1                      ; COMPARISON
        BEQ STOP
        MUL R3,R0,R1                    ; MULTIPLICATION
        MOV R0,R3                       ; Result
        BNE FACT                ; BRANCH TO THE LOOP IF NOT EQUAL
STOP B STOP
        END                             ; Mark end of file
```

Output:
Result can be viewed in Register R0

4. Write a program to add an array of 16 bit numbers and store the 32 bit result in internal RAM

```
        AREA ADDITION, CODE, READONLY
ENTRY                                   ; Mark first instruction to execute
START
        MOV R5, #6                      ; INTIALISE COUNTER TO 6(i.e. N=6)
        MOV R0, #0                      ; INTIALISE SUM TO ZERO
        LDR R1, =VALUE1                 ; LOADS THE ADDRESS OF 1ST VALUE
LOOP
        LDRH R3, [R1], #02              ; READ 16 BIT DATA
        ADD R0, R0, R3                  ; ADD R0=R0+R3
        SUBS R5, R5, #1                 ; DECREMENT COUNTER
        CMP R5, #0
        BNE LOOP                        ; LOOK BACK TILL ARRAY ENDS
        LDR R4, =RESULT                 ; LOADS THE ADDRESS OF RESULT
        STR R0, [R4]                    ; STORES THE RESULT IN R0
STOP B STOP
VALUE1 DCW 0X1111,0X2222,0X3333,0XAAAA,0XBBBB,0XCCCC
                                ; ARRAY OF 16 BIT NUMBERS(N=6)
        AREA DATA2, DATA, READWRITE
                                ; TO STORE RESULT IN GIVEN ADDRESS
```

```
RESULT DCD 0X0
    END                                ; Mark end of file
```

Output:
Result can be viewed in Memory location address specified in R4 and also inregister R0

5. **Write a program to find the square of a number (1 to 10) using look-up table.**

```
            AREA SQUARE, CODE, READONLY

ENTRY                                           ; Mark first instruction to execute
START

            LDR R0, = TABLE1                    ; Load start address of Lookup table
            LDR R2, =  0X40000000
            LDR R1, [R2]                ; Load no whose square is to be find
            MOV R1, R1, LSL #0x2
                        ; Generate address corresponding to square of given no
            ADD R0, R0, R1              ; Load address of element in Lookup table
            LDR R3, [R0]        ; Get square of given no in R3

STOP B STOP

;Lookup table contains Squares of nos from 0 to 10 (in hex)

TABLE1      DCD 0X00000000              ;SQUARE OF 0=0
            DCD 0X00000001              ;SQUARE OF 1=1
            DCD 0X00000004              ;SQUARE OF 2=4
            DCD 0X00000009              ;SQUARE OF 3=9
            DCD 0X00000010              ;SQUARE OF 4=16
            DCD 0X00000019              ;SQUARE OF 5=25
            DCD 0X00000024              ;SQUARE OF 6=36
            DCD 0X00000031              ;SQUARE OF 7=49
            DCD 0X00000040              ;SQUARE OF 8=64
            DCD 0X00000051              ;SQUARE OF 9=81
            DCD 0X00000064              ;SQUARE OF 10=100

            END                                 ; Mark end of file
```

Output:
Enter Input number in memory location specified in Register R2
Result can be viewed in Register R3

6a. Write a program to find the largest number in an array of 32 numbers.

```
        AREA LARGEST, CODE, READONLY

ENTRY                           ;Mark first instruction to execute

START
        MOV R5,#6               ; INTIALISE COUNTER TO 6(i.e. N=7)
        LDR R1,=VALUE1          ; LOADS THE ADDRESS OF FIRST VALUE
        LDR R2,[R1],#4          ; WORD ALIGN T0 ARRAY ELEMENT
LOOP
        LDR R4,[R1],#4          ; WORD ALIGN T0 ARRAY ELEMENT
        CMP R2,R4               ; COMPARE NUMBERS
        BHI LOOP1               ; IF THE 1stNUMBER IS > THEN GOTO LOOP1

        MOV R2,R4               ; IF THE 1stNUMBER IS < THEN MOV
                                ; CONTENT R4 TO R2
LOOP1
        SUBS R5,R5,#1           ; DECREMENT COUNTER
        CMP R5,#0               ; COMPARE COUNTER TO 0
        BNE LOOP                ; LOOP BACK TILL ARRAY ENDS

        LDR R4,=RESULT          ; LOADS THE ADDRESS OF RESULT
        STR R2,[R4]             ; STORES THE RESULT IN R2

STOP B STOP

; ARRAY OF 32 BIT NUMBERS(N=7)

VALUE1
                DCD     0X44444444
                DCD     0X22222222
                DCD     0X11111111
                DCD     0X33333333
                DCD     0XAAAAAAAA
                DCD     0X88888888
                DCD     0X99999999

        AREA DATA2,DATA,READWRITE
                                ; TO STORE RESULT IN GIVEN ADDRESS
RESULT DCD 0X0
        END                     ; Mark end of file
```

Output:

Result can be viewed in Memory location address specified in R4 and also in register R2

6b.Write a program to find the smallest number in an array of 32 numbers.

```
        AREA SMALLEST, CODE, READONLY
ENTRY                               ;Mark first instruction to execute

START
        MOV R5,#6                   ; INTIALISE COUNTER TO 6(i.e. N=7)
        LDR R1,=VALUE1              ; LOADS THE ADDRESS OF FIRST VALUE
        LDR R2,[R1],#4              ; WORD ALIGN T0 ARRAY ELEMENT
LOOP
        LDR R4,[R1],#4              ; WORD ALIGN T0 ARRAY ELEMENT
        CMP R2,R4                   ; COMPARE NUMBERS
        BLS LOOP1                   ; IF THE 1stNUMBER IS < THEN GOTO LOOP1

        MOV R2,R4                   ; IF THE 1stNUMBER IS > THEN MOV
                                     ; CONTENT R4 TO R2
LOOP1
        SUBS R5,R5,#1               ; DECREMENT COUNTER
        CMP R5,#0                   ; COMPARE COUNTER TO 0
        BNE LOOP                    ; LOOP BACK TILL ARRAY ENDS

        LDR R4,=RESULT              ; LOADS THE ADDRESS OF RESULT
        STR R2,[R4]                 ; STORES THE RESULT IN R1

STOP B STOP

; ARRAY OF 32 BIT NUMBERS(N=7)

VALUE1
                DCD    0X44444444
                DCD    0X22222222
                DCD    0X11111111
                DCD    0X22222222
                DCD    0XAAAAAAAA
                DCD    0X88888888
                DCD    0X99999999

        AREA DATA2,DATA,READWRITE
                                    ; TO STORE RESULT IN GIVEN ADDRESS
RESULT DCD 0X0

        END                         ; Mark end of file
```

Output:

Result can be viewed in Memory location address specified in R4 and also in register R2

7a. Write a program to arrange a series of 32 bit numbers in ascending order.

```
            AREA ASCENDING, CODE, READONLY
ENTRY                               ;Mark first instruction to execute
START
            MOV R8,#4               ; INTIALISE COUNTER TO 4(i.e. N=4)
            LDR R2, =CVALUE         ; ADDRESS OF CODE REGION
            LDR R3, =DVALUE         ; ADDRESS OF DATA REGION
LOOP0
            LDR R1, [R2], #4    ; LOADING VALUES FROM CODE REGION
            STR R1,[R3], #4         ; STORING VALUES TO DATA REGION
            SUBS R8, R8, #1         ; DECREMENT COUNTER
            CMP R8,#0               ; COMPARE COUNTER TO 0
            BNE LOOP0               ; LOOP BACK TILL ARRAY ENDS

START1      MOV R5,#3               ; INTIALISE COUNTER TO 3(i.e. N=4)
            MOV R7,#0   ; FLAG TO DENOTE EXCHANGE HAS OCCURED
            LDR R1,=DVALUE          ; LOADS THE ADDRESS OF 1stVALUE

LOOP        LDR R2,[R1],#4          ; WORD ALIGN T0 ARRAY ELEMENT
            LDR R3,[R1]             ; LOAD SECOND NUMBER
            CMP R2,R3               ; COMPARE NUMBERS
            BLT LOOP2           ; IF THE 1stNUMBER IS < THEN GOTO LOOP2
            STR R2,[R1],#-4
            STR R3,[R1]
            MOV R7,#1   ; FLAG DENOTING EXCHANGE HAS TAKEN PLACE
            ADD R1,#4               ; RESTORE THE PTR
LOOP2
            SUBS R5,R5,#1           ; DECREMENT COUNTER
            CMP R5,#0               ; COMPARE COUNTER TO 0
            BNE LOOP                ; LOOP BACK TILL ARRAY ENDS
            CMP R7,#0               ; COMPARING FLAG
            BNE START1
                        ; IF FLAG IS NOT ZERO THEN GO TO START1 LOOP
    STOP B STOP
    ; ARRAY OF 32 BIT NUMBERS(N=4) IN CODE REGION
        CVALUE
            DCD     0X44444444
            DCD     0X11111111
            DCD     0X33333333
            DCD     0X22222222

            AREA DATA1,DATA,READWRITE
;ARRAY OF 32 BIT NUMBERS IN DATA REGION
        DVALUE
            DCD 0X00000000
            END                     ; Mark end of file
```

Output:
Result can be viewed at location DVALUE (stored in R3)

7b. Write a program to arrange a series of 32 bit numbers in descending order.

```
        AREA DESCENDING, CODE, READONLY
ENTRY                               ;Mark first instruction to execute
START
            MOV R8,#4                       ; INTIALISE COUNTER TO 4(i.e. N=4)
            LDR R2,=CVALUE                  ; ADDRESS OF CODE REGION
            LDR R3,=DVALUE                  ; ADDRESS OF DATA REGION
LOOP0
            LDR R1,[R2],#4          ;  LOADING VALUES FROM CODE REGION
            STR R1,[R3],#4          ;  STORING VALUES TO DATA REGION
            SUBS R8,R8,#1                   ; DECREMENT COUNTER
            CMP R8,#0                       ; COMPARE COUNTER TO 0
            BNE LOOP0                       ; LOOP BACK TILL ARRAY ENDS
START1      MOV R5,#3                       ; INTIALISE COUNTER TO 3(i.e. N=4)
            MOV R7,#0     ; FLAG TO DENOTE EXCHANGE HAS OCCURED
            LDR R1,=DVALUE          ; LOADS THE ADDRESS OF FIRST VALUE
LOOP        LDR R2,[R1],#4                  ; WORD ALIGN T0 ARRAY ELEMENT
            LDR R3,[R1]                     ; LOAD SECOND NUMBER
            CMP R2,R3                       ; COMPARE NUMBERS
            BGT LOOP2     ; IF THE 1stNUMBER IS > THEN GOTO LOOP2
            STR R2,[R1],#-4                 ; INTERCHANGE NUMBER R2 & R3
            STR R3,[R1]                     ; INTERCHANGE NUMBER R2 & R3
            MOV R7,#1     ; FLAG DENOTING EXCHANGE HAS TAKEN PLACE
            ADD R1,#4                       ; RESTORE THE PTR
LOOP2       SUBS R5,R5,#1                   ; DECREMENT COUNTER
            CMP R5,#0                       ; COMPARE COUNTER TO 0
            BNE LOOP                        ; LOOP BACK TILL ARRAY ENDS
            CMP R7,#0                       ; COMPARING FLAG
            BNE START1
                          ; IF FLAG IS NOT ZERO THEN GO TO START1 LOOP

STOP B STOP

; ARRAY OF 32 BIT NUMBERS(N=4) IN CODE REGION
CVALUE
        DCD     0X44444444
        DCD     0X11111111
        DCD     0X33333333
        DCD     0X22222222

        AREA DATA1,DATA,READWRITE
                                ; ARRAY OF 32 BIT NUMBERS IN DATA REGION
DVALUE
```

```
            DCD 0X00000000
    END                           ; Mark end of file
```

Output: Result can be viewed at location DVALUE (stored in R3)

8. Write a program to count the number of ones and zeros in two consecutive Memory locations.

```
            AREAONEZERO, CODE, READONLY

ENTRY                                   ;Mark first instruction to execute

START

            MOV R2,#0                   ; COUNTER FOR ONES
            MOV R3,#0                   ; COUNTER FOR ZEROS
            MOV R7,#2                   ; COUNTER TO GET TWO WORDS
            LDR R6,=VALUE               ; LOADS THE ADDRESS OF VALUE

LOOP        MOV R1,#32                  ; 32 BITS COUNTER
            LDR R0,[R6],#4              ; GET THE 32 BIT VALUE

LOOP0       MOVS R0,R0,ROR #1; RIGHT SHIFT TO CHECK CARRY BIT (1's/0's)
            BHI ONES
                  ; IF CARRY BIT IS 1 GOTO ONES BRANCH OTHERWISE NEXT

ZEROS       ADD R3,R3,#1
                        ; IF CARRY BIT IS 0 THEN INC THE COUNTER BY 1(R3)
            B LOOP1                     ; BRANCH TO LOOP1
ONES        ADD R2,R2,#1
                  ; IF CARRY BIT IS 1 THEN INC THE COUNTER BY 1(R2)

LOOP1       SUBS R1,R1,#1            ; COUNTER VALUE DECREMENTED BY 1
            BNE LOOP0          ; IF NOT EQUAL GOTO TO LOOP0 CHECKS 32BIT

            SUBS R7,R7,#1            ; COUNTER VALUE DECREMENTED BY 1
            CMP R7,#0                   ; COMPARE COUNTER R7 TO 0
            BNE LOOP                    ; IF NOT EQUAL GOTO TO LOOP
STOP B STOP

VALUE DCD 0X11111111,0XAA55AA55         ; TWO VALUES IN AN ARRAY

            END                         ; Mark end of file
```

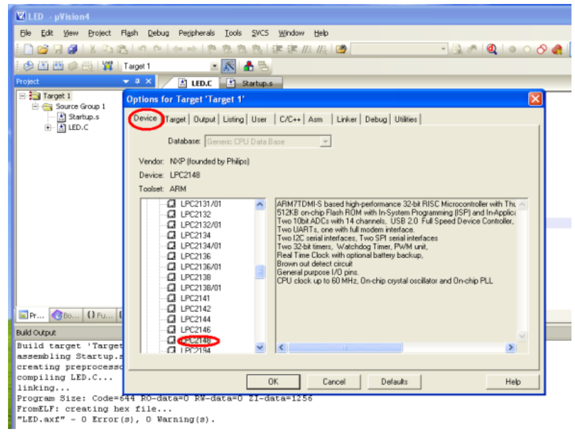
Output:
Result in R2 for ONES & R3 for ZEROS

Hardware Program Execution Steps

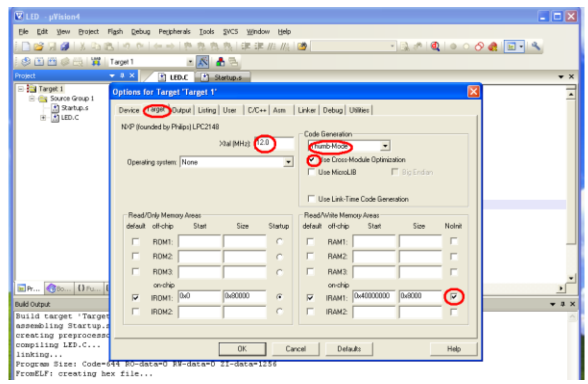
Follow the procedure same as software program from 1 to 15.

<u>HEXfilecreation</u>

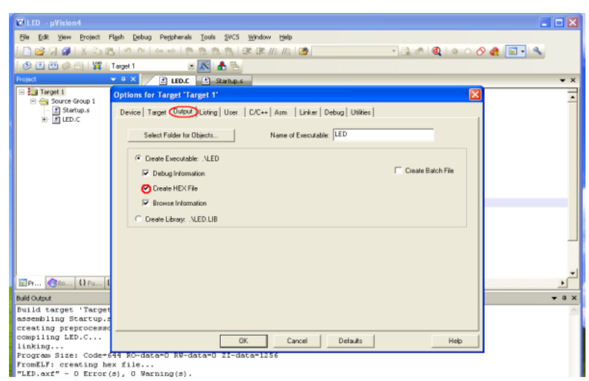
STEP16: Follow the **STEP** upto **16** and in"**Project**" window right click onTarget1 and select **"Option for Target, Target1".** In that window, with in the Device menu we need to select **LPC2148** and click ok.



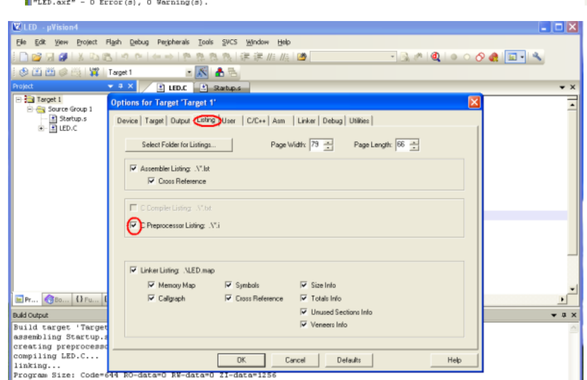
STEP17: Next within the **Target** menu we need to set clock frequency as **12.0** MHz and select **Thumb mode** within the code generation.



STEP 18: Then go to Output click on **create HEX file** and click ok.

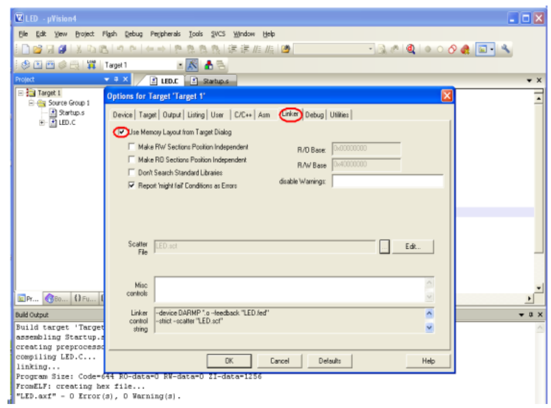


STEP 19: Then within the **Listing** select **C**



preprocessor Listing and click ok.

STEP 20: Finally within the **Linker** click on **use memory layout from target dialog** and click ok.



Repeat Step No 14 and 15 (Build target and Rebuild target) to create Hex File.

<u>FLASH MAGIC</u>

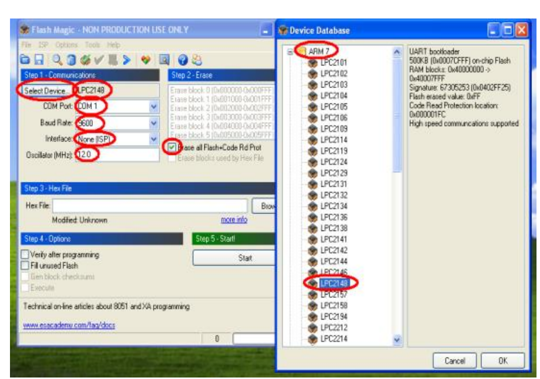
INTRODUCTON

Flash Magic is Windows software from the Embedded Systems Academy that allows easy access to all the ISP features provided by the devices. These features include:

- Erasing the Flash memory (individual blocks or the whole device)
- Programming the Flash memory
- Modifying the Boot Vector and Status Byte
- Reading Flash memory
- Performing a blank check on a section of Flash memory
- Reading the signature bytes
- Reading and writing the security bits
- Direct load of a new baud rate (high speed communications)
- Sending commands to place device in Boot loader mode

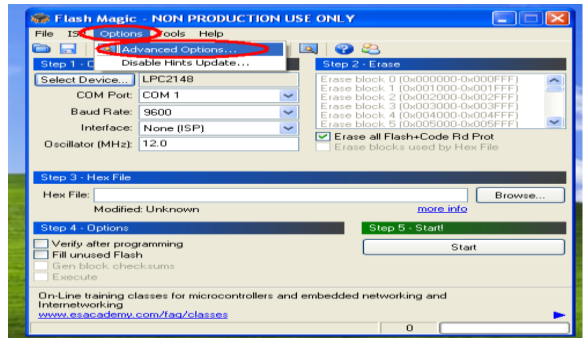
<u>Programming with communication port (COM1)</u>

STEP 21: For programming with communication port first we need to select the device as **LPC2148** within ARM7, comport as **COM1**, baud rate as **9600**,

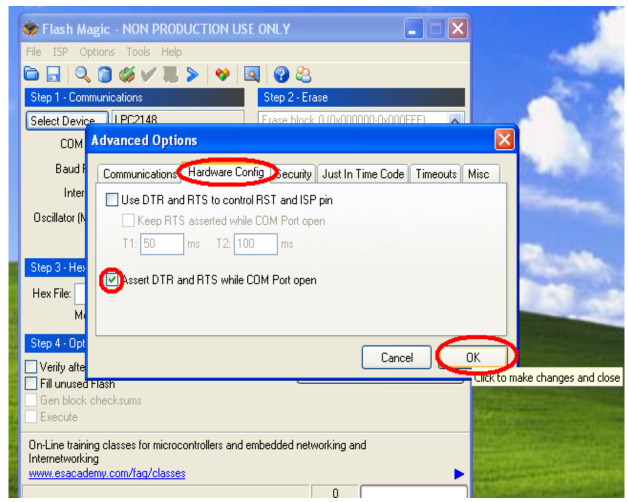


interface **None[ISP],** Oscillator frequency **12.0Mhz**. click on **erase of flash code Rd plot.**

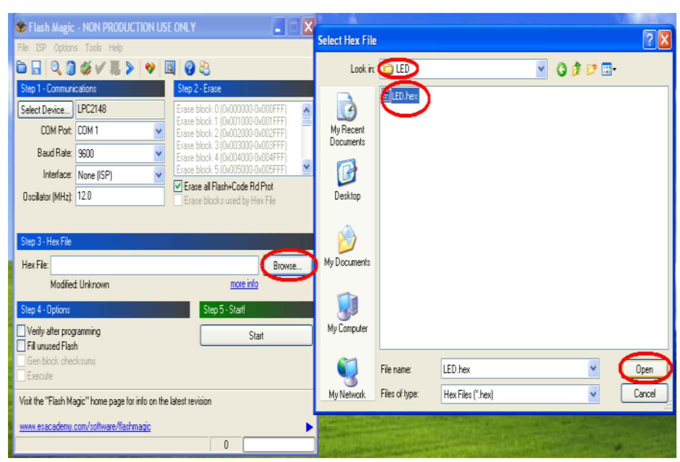
STEP 22: Under the options go to **advanced options.**



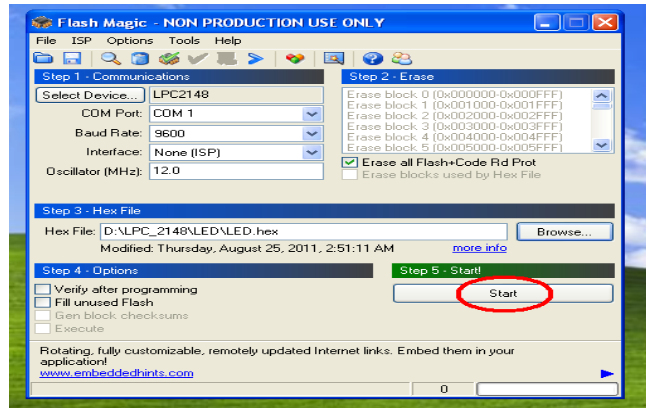
STEP23: Under the Advanced options goto Hardware configuration click on the **Assert DTR and RTS while COM port open** and click ok.



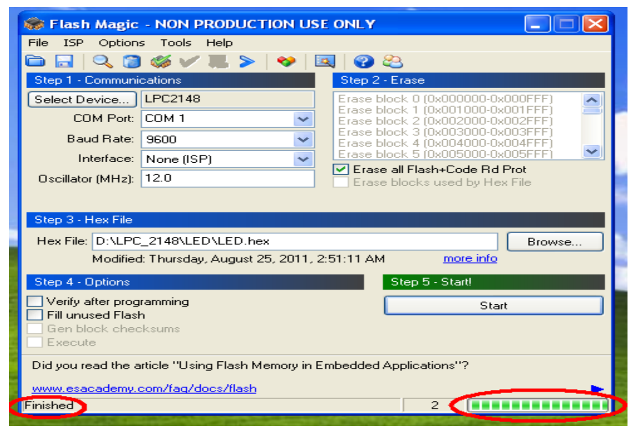
STEP 24: Next browse the file and select the path**.**



STEP 25: After selecting ISP mode and reset at the kit and click on start.



STEP26: After the above steps we can see the finished indication.



PART –B (Hardware Programs)

9. Display "Hello World" message using Internal UART.

```
#include<lpc214x.h>

voiduart_init (void);
unsigned int delay;
unsigned char *ptr;
unsigned char arr[]="HELLO WORLD\r";

int main()
{
        while(1)
        {
                uart_init();
                ptr = arr;
                while(*ptr!='\0')
                {
                        U0THR=*ptr++;
                        while(!(U0LSR & 0x40)== 0x40);
                        for(delay=0;delay<=600;delay++);
                }
                for(delay=0;delay<=60000;delay++);
        }
}
Void uart_init(void)
{
        PINSEL0=0X0000005;                      //select TXD0 and RXD0 lines
        U0LCR = 0X00000083;                 //enable baud rate divisor loading and
        U0DLM = 0X00;                           //select the data format
        U0DLL = 0x13;                           //select baud rate 9600 bps
        U0LCR = 0X00000003;
}
```

10. Interface and Control a DC Motor.

```
#include<lpc214x.h>
void clock_wise(void);
void anti_clock_wise(void);
unsigned int j=0;

int main()
{
        IO0DIR= 0X00000900;
```

```
        IO0SET= 0X00000100;                     //P0.8 should always high.

        while(1)
        {
                clock_wise();
                for(j=0;j<400000;j++);          //delay
                anti_clock_wise();
                for(j=0;j<400000;j++);          //delay
         }                                      //End of while(1)
}                                               //End of Main

void clock_wise(void)
{
        IO0CLR = 0x00000900;            //stop motor and also turn off relay
        for(j=0;j<10000;j++);       //small delay to allow motor to turn off
        IO0SET = 0X00000900;
                        //Selecting the P0.11 line for clockwise and turn on motor
}

void anti_clock_wise(void)
{
        O0CLR = 0X00000900;             //stop motor and also turn off relay
        for(j=0;j<10000;j++);       //small delay to allow motor to turn off
        IO0SET = 0X00000100;    //not selecting the P0.11 line for Anti clockwise
}
```

11. Interface a Stepper motor and rotate it in clockwise and anti-clockwise direction.

```
/* A stepper motor direction is controlled by shifting the voltage across the coils.
Port lines : P0.12 to P0.15 */

#include <LPC21xx.H>
void clock_wise(void);
void anti_clock_wise(void);

unsigned long int var1,var2;
unsigned inti=0,j=0,k=0;

int main(void)
{
        PINSEL0 = 0x00FFFFFF;           //P0.12 to P0.15 GPIo
        IO0DIR |= 0x0000F000;
//P0.12 to P0.15 output. It is used to control the individual control of GPIO pins

        while(1)
        {
```

```
                for(j=0;j<50;j++)            // 20 times in Clock wise Rotation
                clock_wise();

                for(k=0;k<65000;k++);        // Delay to show  anti_clock Rotation

                for(j=0;j<50;j++)       // 20 times in  Anti Clock wise Rotation
                anti_clock_wise();

                for(k=0;k<65000;k++);         // Delay to show clock Rotation

        }                                                       // End of while(1)

}                                                               // End of main

void clock_wise(void)
{
        var1 = 0x00000800;                   //For Clockwise
        for(i=0;i<=3;i++)                    // for A B C D Stepping
        {
                var1 = var1<<1;              //For Clockwise
                IO0PIN = ~var1;              //To access only digital pins

        for(k=0;k<3000;k++);         //for step speed variation
         }

}

void anti_clock_wise(void)
{
        var1 = 0x00010000;                   //For Anticlockwise
        for(i=0;i<=3;i++)                    // for A B C D Stepping
        {
                var1 = var1>>1;              //For Anticlockwise
        IO0PIN = ~var1;
        for(k=0;k<3000;k++);         //for step speed variation
        }
}
```

12.Determine Digital output for a given Analog input using Internal ADC of ARM controller.

```
//10-bit internal ADC
//AIN0 pin is selected
//you can change the channel by changing PINSEL1 and ADCR value

#include <lpc214x.h>
#include <Stdio.h>
```

```
void lcd_init(void);
void wr_cn(void);
void clr_disp(void);
void delay(unsigned int);
void lcd_com(void);
void wr_dn(void);
void lcd_data(void);

unsigned int data_lcd=0;
unsigned int adc_value=0,temp_adc=0,temp1,temp2;
float temp;
char var[15],var1[15];
char *ptr,arr[]= "ADC O/P= ";
char *ptr1,dis[]="A I/P  = ";

#define vol 3.3                                  //Reference voltage
#define fullscale 0x3ff                          //10 bit adc

int main()
{
        PINSEL1 = 0X00040000;                    //AD0.4 pin is selected(P0.25)
        IO0DIR = 0x000000FC;                     //configure o/p lines for lcd

        delay(3200);
        lcd_init();                              //LCD initialization
        delay(3200);
        clr_disp();                              //clear display
        delay(3200);                             //delay

        ptr = dis;
        temp1 = 0x80;               //Display starting addressof first line 1stpos
        lcd_com();
        delay(800);

        while(*ptr!='\0')
        {
                temp1 = *ptr;
                lcd_data();
                ptr ++;
        }

        ptr1 = arr;
        temp1 = 0xC0;         //Display starting address of second line 4thpos
        lcd_com();
        delay(800);
```

```
            while(*ptr1!='\0')
            {
                        temp1 = *ptr1;
                        lcd_data();
                        ptr1 ++;
            }
    //infinite loop
            while(1)
             {                                               //CONTROL register for ADC
                        AD0CR = 0x01200010;         //command register for ADC-AD0.4

                        while(((temp_adc = AD0GDR) &0x80000000)  == 0x00000000);
                                                    //to check the interrupt bit

                        adc_value = AD0GDR;                  //reading the ADC value
                        adc_value>>=6;
                        adc_value&= 0x000003ff;
                        temp = ((float)adc_value * (float)vol)/(float)fullscale;
                        sprintf(var1,"%4.2fV",temp);
                        sprintf(var,"%3x",adc_value);

                        temp1 = 0x89;
                        lcd_com();
                        delay(1200);
                        ptr = var1;

                        while(*ptr!='\0')
            {
                                    temp1=*ptr;
                                    lcd_data();
                        ptr++;
                        }
            lcd_com();
            delay(1200);

                        ptr1 = var;
            while(*ptr1!='\0')
                        {
                        temp1=*ptr1;
                                    lcd_data();
                        ptr1++;
                        }
            }                                                        // end of while(1)
      }                                                              //end of main()

      //lcd initialization
      voidlcd_init()
```

```
{
        temp2=0x30;
        wr_cn();
        delay(800);

        temp2=0x30;
        wr_cn();
        delay(800);

        temp2=0x30;
        wr_cn();
        delay(800);

        temp2=0x20;
        wr_cn();
        delay(800);

        temp1 = 0x28;
        lcd_com();
        delay(800);

        temp1 = 0x0c;
        lcd_com();
        delay(800);

        temp1 = 0x06;
        lcd_com();
        delay(800);

        temp1 = 0x80;
        lcd_com();
        delay(800);
}

void lcd_com(void)
{
        temp2= temp1 & 0xf0;
        wr_cn();
        temp2 = temp1 & 0x0f;
        temp2 = temp2 << 4;
        wr_cn();
        delay(500);
}

// command nibble o/p routine
void wr_cn(void)                                // write command reg
{
```

```
        IO0CLR  = 0x000000FC;                       // clear the port lines.
        IO0SET          = temp2;            // Assign the value to the PORT lines
        IO0CLR  = 0x00000004;               // clear bit  RS = 0
        IO0SET          = 0x00000008;       // E=1
        delay(10);
        IO0CLR  = 0x00000008;
}


// data nibble o/p routine
void wr_dn(void)
{
        IO0CLR  = 0x000000FC;                   // clear the port lines.
        IO0SET = temp2;                     // Assign the value to the PORT lines
        IO0SET = 0x00000004;                    // set bit  RS = 1
        IO0SET = 0x00000008;                    // E=1
        delay(10);
        IO0CLR = 0x00000008;
}

// data o/p routine which also outputs high nibble first
// and lower nibble next
void lcd_data(void)
{
        temp2 = temp1 & 0xf0;
        wr_dn();
        temp2= temp1 & 0x0f;
        temp2= temp2 << 4;
        wr_dn();
        delay(100);
}

void delay(unsigned int r1)
{
        unsignedint r;
        for(r=0;r<r1;r++);
}
void clr_disp(void)
{
        temp1 = 0x01;
        lcd_com();
        delay(500);
}
```

13. Interface a DAC and generate Triangular and Square waveforms.

//Triangular waveforms

```
#include <LPC21xx.h>
int main ()
{
        unsigned long int temp=0x00000000;
        unsigned inti=0;
        IO0DIR=0x00FF0000;
        while(1)
        {
        // output 0 to FE
                for(i=0;i!=0xFF;i++)
                {
                        temp=i;
                        temp = temp << 16;
                        IO0PIN=temp;
                }
        // output FF to 1
                for(i=0xFF; i!=0;i--)
                {
                        temp=i;
                        temp = temp << 16;
                        IO0PIN=temp;
                }
        }                                       //End of while(1)
}                                               //End of main()
```

//Square Waveforms

```
#include <lpc21xx.h>
void delay(void);
int main ()
{
        PINSEL0 = 0x00000000 ;          // Configure P0.0 to P0.15 as GPIO
        PINSEL1 = 0x00000000 ;          // Configure P0.16 to P0.31 as GPIO
        IO0DIR  = 0x00FF0000 ;
        while(1)
        {
                IO0PIN = 0x00000000;
        delay();
        IO0PIN = 0x00FF0000;
        delay();
        }
}
void delay(void)
{
        unsignedinti=0;
```

```
            for(i=0;i<=95000;i++);
    }
```

14. Interface a 4x4 keyboard and display the key code on an LCD.

```
#include<lpc21xx.h>
#include<stdio.h>

/******* FUNCTION PROTOTYPE*******/

void lcd_init(void);
void clr_disp(void);
void lcd_com(void);
void lcd_data(void);
void wr_cn(void);
void wr_dn(void);
void scan(void);
void get_key(void);
void display(void);
void delay(unsigned int);
void init_port(void);

unsigned long int scan_code[16]=
{0x00EE0000,0x00ED0000,0x00EB0000,0x00E70000,
                0x00DE0000,0x00DD0000,0x00DB0000,0x00D70000,
                0x00BE0000,0x00BD0000,0x00BB0000,0x00B70000,
                0x007E0000,0x007D0000,0x007B0000,0x00770000};


unsigned char ASCII_CODE[16]= {'0','1','2','3',
               '4','5','6','7',
               '8','9','A','B',
               'C','D','E','F'};

unsigned char  row,col;

unsigned char temp,flag,i,result,temp1;
unsignedint r,r1;
unsigned long int var,var1,var2,res1,temp2,temp3,temp4;
unsigned char *ptr,disp[] = "4X4 KEYPAD";
unsigned char disp0[] = "KEYPAD TESTING";
unsigned char disp1[] = "KEY = ";
int main()
{
    //      ARMLIB_enableIRQ();
            init_port();                            //port initialisation
```

```
        delay(3200);                    //delay
        lcd_init();                     //lcdintialisation
        delay(3200);                    //delay
        clr_disp();                     //clear display
        delay(500);                     //delay

    //........LCD DISPLAY TEST.........//
        ptr = disp;
        temp1 = 0x81;                   // Display starting address
        lcd_com();
        delay(800);

        while(*ptr!='\0')
        {
            temp1 = *ptr;
            lcd_data();
            ptr ++;
        }

    //........KEYPAD Working.........//
        while(1)
        {
            get_key();
            display();
        }

}                                               //end of main()

voidget_key(void)                       //get the key from the keyboard
{
        unsignedinti;
        flag = 0x00;
        IO1PIN=0x000f0000;
        while(1)
        {
            for(row=0X00;row<0X04;row++)        //Writing one for col's
            {
                if( row == 0X00)
               {
                    temp3=0x00700000;
                }
            else if(row == 0X01)
            {
                temp3=0x00B00000;
                }
                else if(row == 0X02)
                {
```

```
                    temp3=0x00D00000;
                    }
                else if(row == 0X03)
                    {
                        temp3=0x00E00000;
                    }
                    var1 = temp3;
                IO1PIN = var1;          // each time var1 value is put to port1
                    IO1CLR =~var1;
                        // Once again Conforming (clearing all other bits)
                scan();
                delay(100);                     //delay
                if(flag == 0xff)
                break;
            }                                   // end of for
                if(flag == 0xff)
                break;
            }                                           // end of while

            for(i=0;i<16;i++)
            {
                if(scan_code[i] == res1)        //equate the scan_code with res1
                {
                    result =  ASCII_CODE[i];  //same position value of ascii code
                    break;                  //is assigned to result
                }
            }
        }// end of get_key();

        void scan(void)
        {
            unsigned long int t;
            temp2 = IO1PIN;                             // status of port1
            temp2 = temp2 & 0x000F0000;                 // Verifying column key
            if(temp2 != 0x000F0000)                     // Check for Key Press or Not
            {
                delay(1000);         //delay(100)//give debounce delay check again
                temp2 = IO1PIN;
                temp2 = temp2 & 0x000F0000;        //changed condition is same

                if(temp2 != 0x000F0000)             // store the value in res1
            {
                flag = 0xff;
                res1 = temp2;
                t = (temp3 & 0x00F00000);          //Verfying Row Write
                 res1 = res1 | t;               //final scan value is stored in res1
            }
```

```
            else
        {
            flag = 0x00;
        }
        }
}                                                       // end of scan()

void display(void)
{
        ptr = disp0;
        temp1 = 0x80;                   // Display starting address of first line
        lcd_com();

        while(*ptr!='\0')
        {
            temp1 = *ptr;
            lcd_data();
            ptr ++;
        }

        ptr = disp1;
        temp1 = 0xC0;               // Display starting address of second line
        lcd_com();

        while(*ptr!='\0')
        {
            temp1 = *ptr;
        lcd_data();
            ptr ++;
         }
        temp1 = 0xC6;                   //display  address for key value
        lcd_com();
        temp1 = result;
        lcd_data();
}

voidlcd_init (void)
{
        temp = 0x30;
        wr_cn();
        delay(3200);

        temp = 0x30;
        wr_cn();
        delay(3200);

        temp = 0x30;
```

```
            wr_cn();
            delay(3200);

            temp = 0x20;
            wr_cn();
            delay(3200);

// load command for lcd function setting with lcd in 4 bit mode,
// 2 line and 5x7 matrix display

            temp = 0x28;
            lcd_com();
            delay(3200);

// load a command for display on, cursor on and blinking off
            temp1 = 0x0C;
            lcd_com();
            delay(800);

// command for cursor increment after data dump
            temp1 = 0x06;
            lcd_com();
            delay(800);

            temp1 = 0x80;
            lcd_com();
            delay(800);
    }

    voidlcd_data(void)
    {
            temp = temp1 & 0xf0;
            wr_dn();
            temp= temp1 & 0x0f;
            temp= temp << 4;
            wr_dn();
            delay(100);
    }

    voidwr_dn(void)                                 //write data reg
    {
            IO0CLR  = 0x000000FC;                   // clear the port lines.
            IO0SET = temp;                      // Assign the value to the PORT lines
            IO0SET = 0x00000004;                    // set bit  RS = 1
            IO0SET = 0x00000008;                    // E=1
            delay(10);
            IO0CLR = 0x00000008;
```

```
}

voidlcd_com(void)
{
        temp = temp1 & 0xf0;
        wr_cn();
        temp = temp1 & 0x0f;
        temp = temp << 4;
        wr_cn();
        delay(500);
}



voidwr_cn(void)                         //write command reg
{
        IO0CLR  = 0x000000FC;           // clear the port lines.
        IO0SET= temp;                   // Assign the value to the PORT lines
        IO0CLR  = 0x00000004;           // clear bit  RS = 0
        IO0SET = 0x00000008;            // E=1
        delay(10);
        IO0CLR  = 0x00000008;
}

voidclr_disp(void)
{
        // command to clear lcd display
        temp1 = 0x01;
        lcd_com();
        delay(500);
}

void delay(unsigned int r1)
{
        for(r=0;r<r1;r++);
}

voidinit_port()
{
        IO0DIR = 0x000000FC;                    //configure o/p lines for lcd
        IO1DIR = 0XFFF0FFFF;
}
```

15. Demonstrate the use of an external interrupt to toggle an LED On/Off.

```
#include<lpc214x.h>
void Extint0_isr(void) __irq;                          //declaration of ISR
unsigned char int_flag = 0, flag = 0;
int main(void)
{
        IO1DIR |= 0X02000000;
        IO1SET  = 0X02000000;
        PINSEL1=0X00000001;       //Setup P0.16 to alternate function EINT0

        EXTMODE     =0x01;           //edge i.e falling egge trigger and active low
        EXTPOLAR= 0X00;
        VICVectAddr0 = (unsigned long)Extint0_isr;
                                               //Assign the EINT0 ISR function
        VICVectCntl0 = 0x20 | 14;
        //Assign the VIC channel EINT0 to interrupt priority 0
        VICIntEnable |= 0x00004000;                //Enable the EINT0 interrupt

        while(1)                                   //waiting for interrupt to occur
        {
                if(int_flag == 0x01)
                {
                        if(flag == 0)
                        {
                                IO1CLR = 0X02000000;
                                flag = 1;
                        }
                        else if(flag == 1)
                        {
                                IO1SET = 0x02000000;
                                flag = 0;
                        }
                        int_flag = 0x00;
                }
        }
}

void Extint0(void)__irq
{                                         //whenever there is a low level on EINT0
        EXTINT |= 0x01;                            //Clear interrupt
        int_flag = 0x01;
        VICVectAddr = 0;                           //Acknowledge Interrupt
}
```

16. Display the Hex digits 0 to F on a 7-segment LED interface, with an appropriate delay in between

```
\\DISPLAY ARE CONNECTED IN COMMON CATHODE MODE\\

Port0 Connected to data lines of all 7 segment displays
   a
  ----
f|  g |b
 |----|
e|    |c
  ----  . dot
   d
a = P0.16
b = P0.17
c = P0.18
d = P0.19
e = P0.20
f = P0.21
g = P0.22
dot = P0.23


Select lines for four 7 Segments
DIS1    P0.28
DIS2    P0.29
DIS3    P0.30
DIS4    P0.31

Values Corresponding to Alphabets 1, 2, 3 and 4

Unsigned int delay;
Unsigned int Switch count=0;
Unsigned int Disp[16]={0x003F0000, 0x00060000, 0x005B0000, 0x004F0000,
0x00660000,0x006D0000,0x007D0000, 0x00070000,
                        0x007F0000, 0x006F0000, 0x00770000,0x007C0000,
                          0x00390000, 0x005E0000, 0x00790000, 0x00710000};

#define SELDISP1 0x10000000     //P0.28
#define SELDISP2 0x20000000              //P0.29
#define SELDISP3 0x40000000              //P0.30
#define SELDISP4 0x80000000              //P0.31
#define ALLDISP  0xF0000000              //Select all display
#define DATAPORT 0x00FF0000
                //P0.16 to P0.23 Data lines connected to drive 7 Segments
```

```
int main (void)
{
        PINSEL0 = 0x00000000;
        PINSEL1 = 0x00000000;
        IO0DIR  = 0xF0FF0000;
        IO1DIR  = 0x00000000;

        while(1)
        {
                IO0SET |= ALLDISP;                      // select all digits
                IO0CLR = 0x00FF0000;
                                        // clear the data lines to 7-segment displays
                IO0SET = Disp[Switchcount];
                                // get the 7-segment display value from the array

                if(!(IO1PIN & 0x00800000))              // if the key is pressed
                {
                        for(delay=0;delay<100000;delay++)       // delay
                        {}

                        if((IO1PIN & 0x00800000))
                                        // check to see if key has been released
                        {
                                Switchcount++;
                                if(Switchcount == 0x10)
                                // 0 to F has been displayed ?go back to 0
                                {
                                        Switchcount = 0;
                                        IO0CLR =        0xF0FF0000;
                        }
                        }
                }
        }
}
```