Practical Workbook
Introduction to Embedded Systems

1st edition: 2013

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology,
Karachi – 75270, Pakistan
INTRODUCTION

An embedded system is a computer system designed for specific control functions within a larger system, often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts. By contrast, a general-purpose computer, such as a personal computer (PC), is designed to be flexible and to meet a wide range of end-user needs. Embedded systems control many devices in common use today. Embedded systems contain processing cores that are either microcontrollers or digital signal processors (DSP). The key characteristic, however, is being dedicated to handle a particular task.

The workbook is distributed into three parts. Where Part one focuses on basic ASIC application designing. It introduces the use of Keil simulator in first lab with demonstration how to design embedded system programs in second lab session. A few lab sessions then focus on implementing different lab projects on various microcontrollers such as 8051, MSP430, ATmega and Atmel microcontrollers etc.

Four lab sessions in the second part focuses on design for optimization techniques. This highlights techniques to manually minimize and optimize the code to save code space, using code banking for memory expansion, learning relocation of 8051 programs and second part also demonstrates use of code optimization wizard in Keil µvision.

Finally the third part introduces to students, the Texas Instruments TMS320C6416T DSK board using C6416 DSP processor with various other system components including ADCs/ DACs, peripheral interfaces and expansion ports etc. This board is a DSP starter kit for hands on experiments and various student level projects testing and simulation.

After studying and understanding basics of embedded systems and getting know how of some of the microcontrollers and their hardware specifications, the students are encouraged to undertake an embedded system mini project that will enable the students to apply their practical knowledge in designing, testing and simulation of any embedded system.
# CONTENTS

<table>
<thead>
<tr>
<th>Lab Session No.</th>
<th>Object</th>
<th>Page No</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Executing “Hello World” on Bare-Iron C-code</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Building First Embedded System Program</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Interfacing LCD with 8051 Microcontroller</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Creating a simple calculator using MSP430 Microcontroller</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Blinking LED Project utilizing Atmel microcontroller</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>Working with inbuilt Analog to Digital Converter of ATmega Microcontroller</td>
<td>24</td>
</tr>
<tr>
<td>7</td>
<td>Serial interfacing using Atmel microcontroller</td>
<td>29</td>
</tr>
</tbody>
</table>

**Part One: Basic ASIC application Designing**

**Part Two: Designing for Optimization**

| 8               | Using code banking to expand memory for Infineon C501 Microcontroller  | 32      |
| 9               | Learning some efficient coding techniques to manually optimize the code space | 38      |
| 10              | Optimizing the code size via Keil μvision “Code Optimization” wizard  | 43      |
| 11              | Relocating 8051 programs                                              | 49      |

**Part Three: Embedded Systems for Digital Signal Processing**

| 12              | Introduction to DSP Board and Code Composer Studio 3.1                | 53      |
Lab Session 1

OBJECT

Executing “Hello World” using Bare-Iron C-code

THEORY

Introduction:

The lab content of this course begins with some exercises designed for students to gain some familiarity with programming embedded systems at various levels of abstraction with varying amounts of infrastructure support.

Bare Metal Programming

Like assembly, in C you are forced to work directly with memory, pointers, and the underlying arrangement of the bits and bytes that make up your data structures and their layout (packed structures, byte ordering, data types and conversions, etc.) No garbage collection, no forgiveness for buffer overruns, no regular expressions. C is for bare metal programming — which typically requires safe coding practices, tight program design, and a good understanding of the tool chain and the target environment.

But when you want the underlying elements of the data structures to be visible and your algorithms to be working with these bare metal details, C is a great choice for turning out programs that interact with hardware, networks, sensors, and peripherals, and that are small (compiled size), fast (execution time), and have a small footprint (few dependencies, low loading overhead).

With the aid of bare bones compilation modes like gcc -S %1.c -fno-exceptions -s -Os, and of working syntax converters like A2I, whenever you wish to see the kind the assembly code that C is generating for you, you can.

Embrace C for systems level development, and you’ll retain the power of assembly but from a more efficient point of view.

Running first bare-metal program:

In this lab session first a sample “Hello world” program will be executed in KEIL simulator. KEIL is software especially designed for development of low level programs and files and their simulation. The programming can be done in C-Language which is converted into object code (HEX file) through the software. It also has a debugging environment in which programs can be debugged for any errors or warnings.
The GUI for Keil software can be divided into following major segments:

1. **Project Workspace**: Project Workspace is a pane that keeps track of all the files and their association with the currently running project.

2. **Output Window**: Output window comprises of three major portions which are:
   
   i. Build Window
   
   ii. Command Window
   
   iii. Find in Files

   Build Window is the most important window during debugging phase as it displays the results generated during the debugging of the source file.
The HELLO program is a very simple program that prints "Hello World" to the serial port of the 8051. When you can successfully build and run this program, you have used the entire toolset including:

- The uVision integrated development environment.
- The C51 compiler, linker and the uVision simulator/debugger.
- The ASM program is available in one target:
  
  Simulator: uses the uVision simulator for testing

3. Program Editor:
Program Editor is the space where one can write source code in C-language or directly in assembly language.

4. Tool Panel
The tool panel contains all the required shortcut icons that you want to access instantly.

Lab Exercise:

Illustrate the purpose of following program statements in the C-Language code:

```c
#ifndef MONITOR51
  SCON = 0x50;
  TMOD |= 0x20;
  TH1   = 221;
  TR1   = 1;
  TI    = 1;
#endif
```

________________________
________________________
________________________
Lab Session 2

OBJECT

Building First Embedded System Program

THEORY

Create a Project File

To create a new project file select from the µVision menu Project – New Project…. This opens a standard Windows dialog that asks you for the new project file name.

Select a Device

When you create a new project µVision asks you to select a CPU for your project. The Select Device dialog box shows the µVision device database. Just select the microcontroller you use. We are using for our examples the Philips 80C51RD+ controller. This selection sets necessary tool options for the 80C51RD+ device and simplifies in this way the tool configuration.

![Select Device for Target](image)

Fig 2.1: Window for Target device selection

Once you have selected a CPU from the device database you can open the user manuals for that device in the Project Workspace – Books page. These user manuals are part of the Keil Development Tools.
You may create a new source file with the menu option **File – New**. This opens an empty editor window where you can enter your source code. µVision enables the C color syntax highlighting when you save your file with the dialog **File – Save As**… under a filename with the extension *.C.*

**Fig 2.2: Code Editor Window**

Once you have created your source file you can add this file to your project. µVision offers several ways to add source files to a project. For example, you can select the file group in the **Project Workspace – Files** page and click with the right mouse key to open a local menu. The option **Add Files** opens the standard files dialog. Select the **file name** you have just created.
The **STARTUP.A51** file is the startup code for the most 8051 CPU variants. The startup code clears the data memory and initializes hardware and reentrant stack pointers. In addition, some 8051 derivatives require a CPU initialization code that needs to match the configuration of your hardware design. For example, the Philips 8051RD+ offers you on-chip xdata RAM that should be enabled in the startup code. Since you need to modify that file to match your target hardware, you should copy the **STARTUP.A51** file from the folder **C:\KEIL\C51\LIB** to your project folder.

**Group Project Files:**

File group allow you to organize large projects. For the CPU startup code and other system configuration files you may create a own file group in the **Project – Components, Environment, Books...** dialog box. Use the **New (Insert)** button to create a file group named **System Files**. In the project window you may drag and drop the **STARTUP.A51** file to this new file group.
Now, the Project Workspace – Files lists all items of your project. To open a file for editing, double click on the file name in the Project Workspace. You may need to configure the startup STARTUP.A51 in the editor.

Set Tool Options for Target:

μVision lets you set options for your target hardware. The dialog Options for Target opens via the toolbar icon or via the Project - Options for Target menu item. In the Target tab you specify all relevant parameters of your target hardware and the on-chip components of the device you have selected. The following the settings for our example are shown.
The following table describes the options of the Target dialog:

- **Xtal**: Specifies the CPU clock of your device. In most cases this value is identical with the XTAL frequency.

- **Memory Model**: Specifies the C51 Compiler memory model. For starting new applications the default SMALL is a good choice. Refer to "Memory Models and Memory Types" section for a discussion of the various memory models.

- **Allocate On-Chip**: Use multiple DPTR registers Specifies the usage of the on-chip components that are typically enabled in the CPU startup code. If you are using on-chip xdata RAM (XRAM) you should also enable the XRAM access in the STARTUP.A51 file.

- **Off-chip**: Memory Here you specify all external memory areas of the target hardware.

- **Code Banking**: Xdata Banking Specifies the parameters for code and xdata banking. Refer to the "Code Banking" section for more information.

**Build Project and Create a HEX File:**

Typical, the tool settings under Options – Target are all you need to start a new application. You may translate all source files and line the application with a click on the Build Target toolbar icon. When you build an application with syntax errors, µVision will display errors and warning messages in the Output Window – Build page. A double click on a message line opens the source file on the correct location in a µVision editor window.
Once you have successfully generated your application you can start debugging.

Now you may modify existing source code or add new source files to the project. The **Build Target** toolbar button translates only modified or new source files and generates the executable file. µVision maintains a file dependency list and knows all include files used within a source file. Even the tool options are saved in the file dependency list, so that µVision rebuilds files only when needed. With the **Rebuild Target** command, all source files are translated, regardless of modifications.

After you have tested your application, it might be required to create an Intel HEX file and to download the application software into the physical device using a Flash programming utility. µVision creates HEX files with each build process when **Create HEX file** under **Options for Target – Output** is enabled. The **Merge32KHexfile** option is available for Code Banking Applications when you have selected the Extended Linker LX51. You may start your Flash programming utility after the make process when you specify the program under the option **Run User Program #1**.

![Hex file configuration](image)

**Fig 2.4: Hex file configuration**
EXERCISE:

Create a program that prints your name and roll number in a continuous loop on Serial port 2.

OBSERVATIONS:
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________

10
Lab Session 3

OBJECT

Interfacing LCD with 8051 Microcontroller

THEORY

In this lab session students will learn to create an LCD Driver for 8051 Microcontroller. The circuit diagram for the required configuration is given as under. This is a simple circuit for displaying fixed set of statements in a continuous loop. Therefore no input is provided in the circuit. The program as mentioned earlier runs in a continuous loop while printing the predefined statements. The complete circuit diagram is as under;

![Figure 3.1: Circuit diagram for LCD interfacing with microcontroller](image)

Clock/Reset Circuit:

Most microcontrollers have in built oscillator (Clock circuitry) which needs to be connected with external quartz crystal between pins labeled as XTAL1 and XTAL2. In the following circuit snippet the circuit diagram and components required for clock generation are mentioned. The circuit is built from 1.2 MHz Quartz crystal as primary frequency source along with two 33 pico farad capacitors to adjust clock frequency to desired level.

For reset circuit we;
Port 3 Assignment/ Address Latch Assignment:

In the given microcontroller the Address/ Data bits are multiplexed which needs to be demultiplexed before their distribution to external circuit units. For this purpose we use an octal latch to store the address bits in the Address cycle of the bus. The octal latch that is employed in the given circuit is 74373. The latch is enabled only during the address cycle through ALE pin of the microcontroller.

In the circuit snippet it should be noted that only two address bits are provided to the LCD from the octal latch.
**LCD Pin Assignment:**

LM032L LCD character display module is a two line by twenty character display. LCD modules are useful for displaying text information from a system. In large volume applications, the use of custom LCD displays becomes economical. The routines provided should be a good starting point for users whose applications implement a custom LCD.

LM032L LCD character display module can operate in one of two modes. The first (and default) mode is the 4-bit data interface mode. The second is the 8-bit data interface mode. When operating in 4-bit mode, two transfers per character / command are required. 8-bit mode, though easier to implement (less program memory) requires four additional I/O lines. The use of 8-bit mode is strictly a program memory size vs. I/O trade-off. The three most common data interfaces from the microcontroller are:

1. An 8-bit interface.
2. A 4-bit interface, with data transfers on the high nibble of the port.
3. A 4-bit interface, with data transfers on the low nibble of the port.

The LCD module also has three control signals, Enable (E), Read/Write (R_W), and Register Select (RS). The function of each control signal is shown in Table 1.

<table>
<thead>
<tr>
<th>Control Signal</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Causes data/control state to be latched&lt;br&gt;Rising Edge = Latches control state (RS and R_W)&lt;br&gt;Falling Edge = Latches data</td>
</tr>
<tr>
<td>RS</td>
<td>Register Select Control&lt;br&gt;1 = LCD in data mode&lt;br&gt;0 = LCD in command mode</td>
</tr>
<tr>
<td>R_W</td>
<td>Read / Write control&lt;br&gt;1 = LCD to write data&lt;br&gt;0 = LCD to read data</td>
</tr>
</tbody>
</table>

The data pins of LCD (D0-D7) are directly connected with lower eight Address/Data pins (AD0-AD7) of microcontroller. For (E) pin of LCD we have NAND output connected with this pin. The NAND is feed with Read and Write control signals of 8051 Microcontroller. The LSB of Address that is A0 bit is connected with RS control pin of LCD and the A1 address bit is provided to RW control pin of LCD.

The three power pins of the LCD which are V_{SS}, V_{DD}, V_{EE}. The assignment of these pins is as per following table;

<table>
<thead>
<tr>
<th>V_{SS}</th>
<th>Ground</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_{DD}</td>
<td>V_{DD}</td>
</tr>
<tr>
<td>V_{EE}</td>
<td>Disconnected</td>
</tr>
</tbody>
</table>
Output:

The output for the given circuit can be viewed by simulating the configuration. The program as mentioned earlier just prints the predefined statements which can be changed by appropriate modification in the source code. The output shown in LCD should look like the following simulation snippet.
Loading Hex File:

The hex file can be directly loaded into the desired target by double clicking the component that has an internal memory placed in the circuit. It will open a Edit Component window which has a program file option as shown in the snippet below. By loading a hex file into the controller, the controller has program code on its internal ROM. As we start the simulation the component’s program code is fetched and execute from the target file and seems that the code is actually burnt on to the actual device. This feature gives us flexibility in the current environment to execute from any target location.

EXERCISE:

Write down observations regarding the simulation for above mentioned circuit

__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________
__________________________________________________________________________________

15
Lab Session 4

OBJECT

Creating a simple calculator using MSP430 Microcontroller

THEORY

MSP430 Calculator with object oriented ‘C’

The design implements a simple four function floating point calculator using a MSP430F1121 CPU, Alpha numeric LCD Display and Keypad.

The code including the device drivers-is written entirely in CCE ‘C’ and may be fully single stepped.

The MSP430 is a 16-bit, ultra-low power, mixed signal microcontroller. The strengths of the MSP430 lie in its easy-to-learn, C-compiler friendly, 16-bit CPU partnered with flexible low power modes and intelligent, low-power peripherals. As a catalog market product, its versatility is applied across a number of different end-equipments including medical equipment, electricity and sub-metering, and home appliances such as smoke detectors, thermostats, etc... With over 230 parts available, there is likely to be an MSP430 for almost any application.

The circuit configuration and interfacing can be viewed from the snippet shown in Fig 4.1. The LCD interfacing is same as we have seen in the previous lab session. Now we have introduced an input source in the form of numeric keypad. The input for numeric characters (four bits) is feed to the Port 1 of microcontroller and also to the data pins of the LCD. Whereas the input for the operators (four bit) is first latched and then feed to the Port 2 of microcontroller.

Program for the Calculator Application:

The program for simple calculator should be written in normal C syntax. The program then may be converted to HEX file for final implementation on controller. The necessary instructions will be provided by the lab instructor during the lab session.
Fig 4.1: Circuit Diagram for simple calculator
EXERCISE:
Write down the C language code for simple calculator application
Lab Session 5

OBJECT

Blinking LED Project utilizing Atmel microcontroller

THEORY

Introduction:

ATmega16 is an 8-bit high performance microcontroller of Atmel’s Mega AVR family with low power consumption. ATmega16 is based on enhanced RISC (Reduced Instruction Set Computing) architecture with 131 powerful instructions. Most of the instructions execute in one machine cycle. ATmega16 can work on a maximum frequency of 16MHz.

ATmega16 has 16 KB programmable flash memory, static RAM of 1 KB and EEPROM of 512 Bytes. The endurance cycle of flash memory and EEPROM is 10,000 and 100,000, respectively.

ATmega16 is a 40 pin microcontroller. There are 32 I/O (input/output) lines which are divided into four 8-bit ports designated as PORTA, PORTB, PORTC and PORTD.

ATmega16 has various inbuilt peripherals like USART, ADC, Analog Comparator, SPI, JTAG etc. Each I/O pin has an alternative task related to inbuilt peripherals.

LED Interfacing using Atmega microcontroller

This lab session elaborates about setting an output, reading an input and programming simple time depending tasks.

1) Controlling a LED

Plugging in the LED: In the lab we provide simple LEDs with integrated resistors soldered on a connector to be directly plugged onto the board.
The LED will be placed on PORTC5, which is located on the corner of the MC at pin no 28.

**Switching it on**: In the previous paragraph the LED was placed in PORTC5. In the file main.c where the main loop is located the following changes need to be made:
1. Set pin 5 in PORTC to output mode with the Data Direction Register for the port DDRC.
2. Switch pin 5 in PORTC on.
3. Stop the controller.

A program on a microcontroller should never leave its main loop. This normally causes the controller to start from the beginning of the programs. But not all hardware is reset resulting in an undefined state in not properly initialized hardware. To prevent this, normally the controller is kept in an infinite loop at the end of a program.

```c
int main(void) {
    //Set the Data Direction Register to output
    DDRC |= (1<<5);
    //Set the signal to high
    PORTC |= (1<<5);
    //Stop in an infinite loop
    while (1);
}
```

- You can now type `make` to see if the program compiles.
- After connecting the board to power and plugging in the programmer, the binary can be transferred with the command `make prog`.
- After the transfer the LED should now emit light. If it does not, check for possible errors during the transfer on the screen. Or maybe the LED is placed with the wrong polarity. Simply turn it around for a try, wrong polarity does not damage the LED.
Making the LED blink: Now that we know how to switch on a LED, we also want to switch it off again after a certain time. We do this by the simplest way: a delay function. The AVR library provides a very accurate delay function. Make sure your file has the line `#include <util/delay.h>` to include the delay functions.

Now the program looks like this:
```c
#include <util/delay.h>

int main(void) {
    //Set the Data Direction Register to output
    DDRC |= (1<<5);
    while (1) {
        //Set the signal to high
        PORTC |= (1<<5);
        //wait 0.5 sec
        _delay_ms(500);
        //Set the signal to low
        PORTC &= ~(1<<5);
        //wait 0.5 sec
        _delay_ms(500);
    }
}
```

After compiling and transferring the program with `make prog` the led should blink in a frequency of 1 sec.

If it is blinking faster or slower then you maybe have forgotten to fuse the controller. Try making fuse and see if it changes.

This program does what it is supposed to, but it can be written shorter and with a reduction of redundant code. First the bit is set and later cleared; this can also be done with the toggle function which is the equivalent of the XOR operator.

```c
#include <util/delay.h>

int main(void) {
    //Set the Data Direction Register to output
    DDRC |= (1<<5);
    while (1) {
        //Toggle the signal
        PORTC ^= (1<<5);
        //wait 0.5 sec
        _delay_ms(500);
    }
}
```

Making the LED blink (advanced): The previous program is simple but the delay function blocks the whole main loop from working on other tasks. It seems wise to find a method of doing actions in a certain time but without blocking the controller. Such situations are normally resolved using a state machine.

In the basis-code a function called `getMsTimer()` is provided. It returns the time in milliseconds.
(1/1000s) since resetting the controller. Beware that the return value of this function is a uint32 t, when storing it.

```c
#define delay 500
int main(void) {
init();
//Set the Data Direction Register to output
DDRC |= (1<<5);
uint32_t next_ms_timer=getMsTimer()+delay;
while (1) {
//Is it time to do smth.?
if (getMsTimer()>next_ms_timer) {
//Toggle the signal
PORTC ^= (1<<5);
//don't forget to set the next "alarm"
next_ms_timer+=delay;
}
//plenty of free time to do other stuff
}
}
```

2) Reading an input

If the bit in the DDR (Data Direction Register) for the corresponding pin is 0, it is configured as input. A voltage provided at that pin is interpreted as either 0 or 1 and stored in the PIN-Register (e.g. PIN). In the following example a switch is placed on pin C3. While the switch is pushed, the LED (Still placed at C5) is turned off.

```c
#define delay 500
int main(void) {
init();
//Set the Data Direction Register to output
DDRC |= (1<<5);
//Set the Data Direction Register for the switch to input
DDRC &= ~(1<<3);
while (1) {
if (PINC & (1<<3))
PORT|=(1<<5);
else
PORT&=~(1<<5);
}
}
```

**Detecting an edge**: Most of the time it is required to detect a change of a pin. This is done by storing the old value and comparing it to the current one.

```c
#define delay 500
int main(void) {
init();
//Set the Data Direction Register to output
DDRC |= (1<<5);
```
init();
//Set the Data Direction Register for the LED to output
DDRC |= (1<<5);
//Set the Data Direction Register for the switch to input
DDRC &= ~(1<<3);
//save old state
uint8_t oldc=PINC;
while (1) {
  if ((oldc & (1<<3)) != (PINC & (1<<3))) {
    //Switch LED
    PORTC^=(1<<5);
    //save new state
    oldc=PINC;
  }
}

Figure 12.3: Differently connected switches: (a) between 5V and GND, (b) between pin and GND with an external pull-up resistor, (c) as (b) but with an internal pull-up

**Providing a clean signal:** Some words on a well-defined signal for the input should be said. The input stage of the controller has an intrinsic capacitor that keeps its electric charge for a certain time. If you connect a switch, it should always switch the pin between 5V and GND (0V) to allow it to get properly (dis-)charged. This is shown on Fig 12.3 (a).

If a switch cannot change between two connectors but just opens and closes a single line, a so called pull-up resistor must be used. This can be an external resistor with a value between 100K and 1M (See Fig 12.3 (b)).

The most elegant way is to use the internal pull-up already installed inside the Atmega. To activate them the PORT is set to 1, while the DDR remains set to 0.

```
DDRC &= ~(1<<3);
PORTC = (1<<3);
if (PINC & (1<<3)) {
  //do smth.
}
```

Be aware of the fact that by using a pull-up the logic of the switch is inverted!
Lab Session 6

OBJECT

Working with inbuilt Analog to Digital Converter of ATmega Microcontroller

THEORY

Introduction:

Microcontroller understands only digital language. However, the inputs available from the environment to the microcontroller are mostly analog in nature, i.e., they vary continuously with time. In order to understand the inputs by the digital processor, a device called Analog to Digital Converter (ADC) is used. As the name suggests this peripheral gathers the analog information supplied from the environment and converts it to the controller understandable digital format, microcontroller then processes the information and provides the desired result at the output end.

![Analog Signal](image1.png) → [ADC] → ![Digital Signal](image2.png)

**Figure 13.1: Analog to Digital Conversion**

ATmega16 has an inbuilt 10 bit, 8-channel ADC system. Some of the basic features of Armega16 ADC are:

- 8 Channels
- 10-bit Resolution
- Input voltage range of 0 to Vcc
- Selectable 2.56V of internal Reference voltage source
- AREF pin for External Reference voltage
- ADC Conversion Complete Interrupt

ADC channels in Atmega16 are multiplexed with PORTA and use the common pins (pin33 to pin40) with PORTA. ADC system of Atmega16 microcontroller consists of following pins:

- **ADC0-ADC7**: 8 Channels from Pin 40 to Pin 33 of Atmega16 ADC peripheral.
- **AREF**: Pin32 of Atmega16 microcontroller, the voltage on AREF pin acts as the reference voltage for ADC conversion, reference voltage is always less than or equal to the supply voltage, i.e., Vcc.
- **AVCC**: Pin30, this pin is the supply voltage pin for using PORTA and the ADC; AVCC pin must be connected to Vcc (microcontroller supply voltage) to use PORTA and ADC. It should be External reference voltage source can be used at AREF pin. However, Atmega16 also has internal reference voltage options of 2.56V and Vref = Vcc.

The figure below shows the pin configuration for ADC system of Atmega16 microcontroller.

![Figure 13.2: Pin Configuration for ADC System](image)

**Configuring ADC Registers:**
To use the ADC peripheral of Atmega16, certain registers need to be configured.

1) **ADMUX (ADC Multiplexer And Selection Register)**

   ![ADMUX Register](image)

   **REFS[0:1] bits** determine the source of reference voltage whether it is internal or the external voltage source connected to AREF pin. **MUX [4:0] bits** are used to select between the channels which will
provide data to ADC for conversion. ADLAR bit when set to 1 gives the left adjusted result in data registers ADCH and ADCL.

2) **ADCSRA (ADC Control and Status Register)**

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADEN</td>
<td>ADSC</td>
<td>ADATE</td>
<td>ADIF</td>
<td>ADIE</td>
<td>ADPS2</td>
<td>ADPS1</td>
<td>ADPS0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**ADC Prescaler Sel. Bits**

- 000 → 2
- 001 → 2
- 010 → 4
- 011 → 8
- 100 → 16
- 101 → 32
- 110 → 64
- 111 → 128

**ADEN**: ADC Enable bit, this bit must be set to 1 for turning ADC on.

**ADSC**: ADC Start Conversion bit, this bit is set to 1 to start ADC conversion, as soon as conversion is completed this bit is set back to 0 by the hardware.

**ADATE**: ADC Auto Trigger Enable, this bit is set to 1 to enable auto triggering of ADC conversion.

**ADIF**: ADC Interrupt Flag, this bit is set to 1 when ADC conversion gets complete.

**ADIE**: ADC Interrupt Enable, this bit is set to 1 if we want to activate the ADC conversion complete interrupt.

**ADPS[0:2]**: ADC Prescaler bits, these bits are used to set the ADC clock frequency, the configuration of these bits determine the division factor by which the microcontroller clock frequency is divided to get the ADC clock frequency. The figure above shows the prescaler bit values for respective division factor.

\[
\text{ADC Clock Frequency} = \frac{\text{XTAL Frequency}}{\text{Prescaler}}
\]

**Example**: XTAL Freq = 12 MHz

- Prescaler = 32
- ADC Clock Freq. = \(\frac{12000000}{32}\) = 375 KHz

The ADC clock frequency must lie somewhere between 50 KHz to 200 KHz.

3) **ADCH & ADCL (ADC Data Registers)**
When the ADC conversion is complete the data is stored in these two registers. The data configuration depends on the ADLAR bit value of ADMUX register. If ADLAR=0, data is right adjusted and if ADLAR=1, data is left adjusted. Always read ADCL first and then ADCH. In cases where the 8-bit precision is enough set the ADLAR bit to 1 to left adjust the data and read only the ADCH data register.

When ADLAR = 0,

<table>
<thead>
<tr>
<th>ADCH</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>ADC9</th>
<th>ADC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCL</td>
<td>ADC7</td>
<td>ADC6</td>
<td>ADC5</td>
<td>ADC4</td>
<td>ADC3</td>
<td>ADC2</td>
<td>ADC1</td>
</tr>
</tbody>
</table>

When ADLAR = 1,

<table>
<thead>
<tr>
<th>ADCH</th>
<th>ADC9</th>
<th>ADC8</th>
<th>ADC7</th>
<th>ADC6</th>
<th>ADC5</th>
<th>ADC4</th>
<th>ADC3</th>
<th>ADC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCL</td>
<td>ADC1</td>
<td>ADC0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Circuit Interconnections**

Connect the circuit as shown in the circuit diagram below. A ceramic capacitor 104 is connected in between AVcc (pin 30) and Aref (pin 32). AVcc (pin 30) is connected to external supply +5V.

![Circuit Diagram for ADC System](image-url)
To interface analog device with AVR microcontroller, follow the following steps for programming it.

**Step1:** To initialize ADC
- Set the value in ADMUX register according to the ADC channel and the reference voltage.
- Set the Prescaler bits accordingly in ADCSRA register.
- Set the ADEN bit to enable the ADC.

```c
void ADC_init(void) // Initialization of ADC
{
    ADMUX=(1<<REFS0); // AVcc with external capacitor at AREF
    ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0); // Enable ADC and set Prescaler division factor as 128
}
```

**Step2:** To read the analog value
- Put the channel value in ADMUX.
- Start the conversion by setting the ADSC bit.
- Monitor the ADIF bit for conversion complete.
- Clear the conversion bit ADIF. By writing it 1.
- Digital converted result is now available in ADCH and ADCL registers.

```c
unsigned int ADC_read(unsigned char ch)
{
    ch= ch & 0b00000111; // channel must be b/w 0 to 7
    ADMUX |= ch; // selecting channel
    ADCSRA|=(1<<ADSC); // start conversion
    while(!(ADCSRA & (1<<ADIF))); // waiting for ADIF, conversion complete
    ADCSRA|=(1<<ADIF); // clearing of ADIF, it is done by writing 1 to it
    return (ADC);
}
```
Lab Session 7

OBJECT

Serial interfacing using Atmel microcontroller

THEORY

Serial connection (UART):
The Atmega controller has a so called Universal Asynchronous Receiver Transmitter (UART) which can be used to communicate with any other RS-232 device like a PC. The only hardware required is a signal converter that connects the 0 …… 5V signal of the microcontroller to the -12 …..12V of the PC. We provide such converters in the lab.

When programming UART one should always be aware of two pitfalls:

1. Buffer size: The hardware buffer for input and output on the Controller only hold 1 byte. That means:
   a) If the program does not read fast enough, it loses bytes.
   b) It is only possible to send one byte at a time.

To prevent data loss and blocking calls for sending it is possible to use interrupts. Those functions are currently not implemented in the basis.

2. Speed: The baud (bandwidth) has to be generated by the controller’s oscillator. An internal prescaler is applied to generate the baud. That limits the baud usable with a 16MHz oscillator to 38.400kbit/s.

UART in basis:
First the baud has to be specified in uart.h. After initializing with uartInit() the following functions can be used:

- uart putc(unsigned char c) sends the char stored in c. It blocks if the send buffer is not empty.
- uart puts(char *s) sends a string (terminated with '0') stored in s.
- uart puts pgm (const prog_char * str) sends a string like uart puts(). The only difference is, that this string is stored in ash memory and therefor does not waste valueable SRAM.
- uart puti (int16_t i) sends an integer. Be careful that cannot be longer than 16 bit.
- unsigned char uart getc() receives a char. It blocks until one is in the in buffer.
- uart data waiting() determines if the input buffer stores a valid byte. This can be used as a test before calling uart getc() because uart data waiting() is non-blocking.

Lab Task:
In this lab session you are required to implement a serial interface of Atmel microcontroller with PC. The circuit diagram is as under;
**Fig: 2.1** Circuit diagram for implementing Serial Interface on Atmel

**Example:**

```c
int main(void) {
    // This string wastes RAM, because it's static and could be stored in flash
    uartInit();
    // This string is stored in program memory
    uart_puts("Hello World\n\r");
    uart_puts_pgm(PSTR("Hello World!\n\r");
    if (uart_data_waiting())
        uart_puts_pgm(PSTR("There is already data in the input buffer!\r\n");
    while (1)
    {
        // get a character (blocking)
        uint8_t c = uart_getc();
        // send it back, aka echo
        uart_putc(c);
    }
}
```
The terminal:

On the side of the PC a terminal program is needed to directly communicate with the controller. There is for example minicom which is old and text based or gtkterm.

Settings:

- Baud as you selected in uart.h, i.e. 1200, 9600, 38400 . . .
- 8 bit, no parity, 1 stop bit, a.k.a. 8N1
- No hardware flow control (RTS/CTS)
- No software ow control (Xon/Xoff)

EXERCISE:

With consultation to your instructor implement serial interface with Atmel microcontroller on breadboard and attach simulation/implementation results for the said task.
Lab Session 8

OBJECT

Using code banking to expand memory for Infineon C501 Microcontroller

THEORY

Introduction:

The following topics build on the guidelines, but are more on an architecture level. They show how to design C code for the 8051 to get further reductions in code size and CPU cycles. A standard 8051 device has an address range of 64 KBytes for code space. To expand program code beyond this 64KB limit, the Keil 8051 tools support code banking. This technique lets you manage one common area and 32 banks of up to 64 Kbytes each for a total of 2 Mbytes of bank-switched memory.

The objective of this lab session is to implement bank switching that shows how you can run (simulate) code executing in several code banks.

Creating a Code Banked Project

Keil software implements all the features to support applications based on banked memory. A few aspects of the hardware design need to be specified to the linker so it can create a code banking program that works on the target system.

Implementation of a code banked project with KeilμVision resides in three main operations:

- The Keil Software configuration for banked application support.
- The L51_BANK.A51 and STARTUP.A51 source code files configuration to fit the Infineon C501 architecture.
- Affect a bank number to each source file.

1.1 Keil Software Configuration

To get KeilμVision ready for use for a banked project, only perform the following steps:

- Run KeilμVision.
- Select ‘Project’ menu and ‘New Project’.
- Enter the project name “codebank”
- Select the target microcontroller: Infineon -> C501
- Then press ‘OK’.
A pop-up window appears:

- Press ‘YES’ to include the Standard 8051 Startup Code to your project.
- Define the project options:
  - Choose ‘Project’ menu.
  - Select ‘Options for Target’ Item.
  - In the ‘Target’ tab, activate ‘Code Banking’.
  - Then select 4 as ‘Banks’ number.
  - And define the ‘Bank Area’ as 0x8000 – 0xFFFF.
- Then Press ‘OK’.
1.2 Source File Bank Number Affectation

Once the project created you need to create your source code files. Here in the example we have created four files named `croot.c`, `cbank0.c`, `cbank1.c` and `cbank2.c`.

- Add all the files to the project.
  - Right click on the ‘Source Group 1’ item of your project.
  - Select ‘Add Files to...’
  - Add all your source code files.

Each source file from the project shall be assigned to a bank number. The default configurations for all source files are mapped on the common area. The following steps show how to affect another memory bank to the source file.

- Select a source file.
- Right click on this file.
- Select the “Options for file ‘myfile’” item.
- Select the bank number to affect to the source file.
  - The files mapped to bank 0 refer to the first upper bank (first bank from address 0x8000 to 0xFFFF)
  - The files mapped to bank 1 refer to the second upper bank (second bank from address 0x8000 to 0xFFFF)
  - The files mapped to bank 2 refer to the third upper bank (third bank from address 0x8000 to 0xFFFF)
- Then select ‘OK’.
1.3 L51_BANK.A51 and STARTUP.A51 source files configuration

With KeilμVision software, a code banking application must include the assembly files L51_BANK.A51 and STARTUP.A51. These source modules contain the code that is invoked to switch code banks and the initial context. The L51_BANK.A51 and STARTUP.A51 are located in the folder ‘\Keil\C51\LIB’ under your Keilinstall path. Copy them to your project environment.

1.4 Source Code

Add the following lines of code to the source file created:

- Source code for the file croot.c:

```c
#include <stdio.h>
#include <reg51.h>
extern void func0(void);
extern void func1(void);

void main(void) {
    /* INITIALIZE SERIAL INTERFACE TO 2400 BAUD @12MHz */
    SCON = 0x52;    /* SCON */
    TMOD = 0x20;    /* TMOD */
    TCON = 0x69;    /* TCON */
    TH1 = 0xf3;    /* TH1 */
}```
printf("MAIN PROGRAM CALLS A FUNCTION IN BANK 0 \n");
func0();
printf("MAIN PROGRAM CALLS A FUNCTION IN BANK 1 \n");
func1();

while(1);
}

• Source code for the file cbank0.c:

#include <stdio.h>

extern void func2(void);

void func0(void) {
printf("FUNCTION IN BANK 0 CALLS A FUNCTION IN BANK 2 \n");
func2();
}

• Source code for the file cbank1.c:

#include <stdio.h>

extern void func2(void);

void func1(void) {
printf("FUNCTION IN BANK 1 CALLS A FUNCTION IN BANK 2 \n");
func2();
}

• Source code for the file cbank2.c:

#include <stdio.h>

void func2(void) {
printf("THIS IS A FUNCTION IN BANK 2! \n");
}

Right click on ‘Target1’ and select the option ‘Build target’ or simply press F7. The Build Output should look like the following:
Now debug the application by selecting **Debug ->Start/Stop Debug Session**. After the Disassembly for the project is generated, select the option ‘**Run**’ from the Debug menu and then select **View->Serial Windows->UART # 1** to observe the output.

**EXERCISE:**
Create a code banking application comprised of two banks. There has to be two c source files one for each bank, each file containing a string array. Print the contents of arrays through main source file.

**OBSERVATIONS:**
Lab Session 9

OBJECT

Learning some efficient coding techniques to manually optimize the code space

THEORY

Many configuration parameters have influence on the code quality of your 8051 application. Although, for most applications the default tool setting generates very good code, but in this lab session we will see some guidelines build on an architecture level. They show how to design C code for the 8051 to get further reductions in code size and CPU cycles.

1) Switch vs. If-Else Statements

When making a multipath decision based on the state of a variable (for example, when implementing a state machine) you can use either a series of if-else if-else statements or a switch-case construct. To test which method is less costly, first examine the switch option. Follow the steps below to create the project and c source file:

Steps:

- Run Keil μVision.
- Select ‘Project’ menu and ‘New Project’.
- Enter the project name “codeoptimize”
- Select the target microcontroller.
- Then press ‘OK’
- A pop-up window appears.
  - Press ‘YES’ to include the Standard 8051 Startup Code to your project.
- Once the project created you need to create your source code file. Here in the example we have created a file named cfile.c.
- Add the file to the project.
  - Right click on the ‘Source Group 1’ item of your project.
  - Select ‘Add Files to...’
  - Add your source code file.
- Add the following lines of code to the source file created:

/* basic state machine
note that C coding best practices require having:
- a break statement at the end of each case, and
- a default case */

#include <stdio.h>
#include <reg51.h>

void main (void)
{
    char state = 0;
    switch (state)
Right click on ‘Target1’ and select the option ‘Build target’ or simply press F7. The Build Output should look like the following:

Observe the code size using Switch statement. Now, examine the if-else construct, replace the existing code with the following one:

```c
#include <stdio.h>
#include <reg51.h>

void main (void) {
    char state = 0;
    /* basic state machine */
    if (state == 0)
    {
        state++;
    }
    else if (state == 1)
    {
        state--;
    }
    else
    {
        state = 0;
    }
}
```
The code size has increased in this case. Hence, the general rule for code of any complexity is to use the switch statement.

2) Passing Structures

In C, it is possible to pass a `structure` to a function. You can either pass the structure directly (if it is small) or pass a pointer to a structure. The following is a simple example of passing a structure directly. Add the given code in c source file:

```c
#include <stdio.h>
#include <reg51.h>

/* structure definition and instance */
struct myStruct
{
    char x, y;
} testvar = {1, 2};

/* function with structure passed in directly, returns the sum of the structure’s members */
char doAdd(struct myStruct str)
{
    return str.x + str.y;
}

void main()
{
    /* test call for the above function */
    char testvar2 = doAdd(testvar);
    /* structure passed directly */
}
```
Note the code size in this case. Now we will examine the alternate option, passing by reference (passing a pointer to the structure). Replace the existing code with the one below:

```
#include <stdio.h>
#include <reg51.h>

/* function with structure passed in by reference, returns the sum of the structure’s members */

struct myStruct
{
    char x, y;
} testvar = {1, 2};

chardoAdd(struct myStruct str)
{
    return str.x + str.y;
}

void main (void)
{
    /* test call for the above function */
    char testvar2 = doAdd(testvar);
    /* structure passed by reference */
}
```

See the significant decrease in code size. The lesson from this topic is simple: try to avoid passing structures to functions, regardless of method. Consider passing structure members instead, or just make the structures static or even global.
EXERCISE:

You have seen the impact of using switch, if-else and passing structures by value and reference. Keil c compiler supports two types of pointers, generic and memory-specific. Study and create a program using pointers and observe the impact on code size using these two types of pointers.

OBSERVATIONS:
Lab Session 10

OBJECT

Optimizing the code size via Keil µvision “Code Optimization” wizard

THEORY

Many configuration parameters have influence on the code quality of your 8051 application. Although, for most applications the default tool setting generates very good code, you should be aware of the parameters that improve code density and execution speed.

We will now explore the available options in this wizard.

Code Optimization

1) **Level**: Sets the optimization level the compiler uses when generating object code.

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0     | **Constant Folding**: The compiler performs calculations that reduce expressions to numeric constants, where possible. This includes calculations of run-time addresses.  
**Simple Access Optimizing**: The compiler optimizes access of internal data and |
1. **Jump Optimizing**: The compiler always extends jumps to the final target. Jumps to jumps are eliminated.

2. **Dead Code Elimination**: Unused code fragments and artifacts are eliminated.
   **Jump Negation**: Conditional jumps are closely examined to see if they can be streamlined or eliminated by the inversion of the test logic.

3. **Data Overlaying**: Data and bit segments suitable for static overlay are identified and internally marked. The BL51 Linker/Locator has the capability, through global data flow analysis, of selecting segments which can then be overlaid.

4. **Peephole Optimizing**: Redundant MOV instructions are removed. This includes unnecessary loading of objects from the memory as well as load operations with constants. Complex operations are replaced by simple operations when memory space or execution time can be saved.
   **Register Variables**: Automatic variables and function arguments are located in registers when possible. Reservation of data memory for these variables is omitted.
   **Extended Access Optimizing**: Variables from the IDATA, XDATA, PDATA and CODE areas are directly included in operations. Intermediate registers are frequently unnecessary.
   **Local Common Sub expression Elimination**: The compiler detects multiple uses of the same expression or sub expression. The result of the first expression is saved and reused when possible. Superfluous expression calculations are eliminated from the code.
   **Case/Switch Optimizing**: Code involving switch and case statements is optimized using jump tables or jump strings.
   **Simple Loop Optimizing**: Program loops that fill a memory range with a constant are converted and optimized.

5. **Global Common Subexpression Elimination**: Identical subexpressions within a function are calculated only once when possible. The intermediate result is stored in a register and reused.

6. **Loop Rotation**: Program loops are rotated if the resulting program code is faster and more efficient. The loop expression of for and while loops is evaluated once at the top of the loop and then at the bottom of the loop. This optimization generates more code but speeds up execution.

7. **Extended Index Access Optimizing**: DPTR is used for register variables where appropriate. Pointer and array access are optimized for both execution speed and code size.

8. **Common Tail Merging**: When there are multiple calls to a single function, some of the setup code can be reused, thereby reducing program size.

9. **Common Block Subroutines**: Recurring instruction sequences are detected and converted into subroutines. The Cx51 Compiler rearranges code to obtain larger recurring sequences.

---

**Note**: Each higher optimization level contains all of the characteristics of the preceding lower optimization level. For example, OPTIMIZE level 9 includes all optimizations of levels 0 to 8.
2) **Emphasis**: Specifies whether the compiler should optimize for code size or execution speed.

3) **Global Register Coloring**: Enables application register optimization. When set, the C51 compiler knows the registers used by external functions, and µVision performs an iterative re-translation of C source files to improve register allocation.

4) **Linker Code Packing (max. AJMP / ACALL)**: Instructs the Compiler to include information in the object file for the linker-level program optimizations. When enabled, the LX51 linker/locater performs this optimization.

5) **Don’t use absolute register accesses**: Disables absolute register addressing for registers R0 through R7 and allows function calls across register banks. The code will be slightly longer, because the C compiler cannot use ARx symbols, for example in PUSH or POP instructions, and has to insert replacement code. However, the code will be independent of the selected register bank.

Now we will create a simple application containing two functions `o1Test1()` and `o1Test2()`. The function will be called in `main()`. We will see the effect of selecting different optimization levels for on the code size.

**Steps:**

- Run Keil µVision.
- Select ‘Project’ menu and ‘New Project’.
- Enter the project name “codeoptimize”.
- Select the target microcontroller.
- Then press ‘OK’.
  - A pop-up window appears.
- Press ‘YES’ to include the Standard 8051 Startup Code to your project.
- Once the project created you need to create your source code file. Here in the example we have created a file named `cfile.c`.
  - Add the file to the project.
- Right click on the ‘Source Group 1’ item of your project.
  - Select ‘Add Files to...’
  - Add your source code file.

Add the following lines of code to the source file created:

```c
#include <stdio.h>
#include <reg51.h>

void o1Test1()
{
    unsigned int x = 1;
    unsigned int y = x + 2;
    x = y - 1;
}

void o1Test2()
{
    unsigned int a = 3;
}
```

45
unsigned int b = a + 5;
    a = b - 1;
}

void main()
{
    olTest1();
    olTest2();
}

Right click on ‘Target1’ and select the option ‘Options for Target 1’, then click on the tab C51. You will see the Keil µvision’s code optimization wizard. Now select the desired Level of optimization, say Level 2: Data overlaying, then click ‘OK’.

Finally build the target, right click on ‘Target1’ and select the option ‘Build target’ or simply press F7. The Build Output should look like the following:

Observe the code size when level of optimization is set to ‘2’.

Now change the level of optimization, make it 9: Common Block Subroutines, and observe the significant reduction in code size.
You may change the optimizer level on a function-by-function basis as needed. For example:

```c
#include <stdio.h>
#include <reg51.h>

#pragma OT(9) /* setting optimization level 9 for function
void olTest1() */
{
    unsigned int x = 1;
    unsigned int y = x + 2;
    x = y - 1;
}

#pragma OT(5) /* setting optimization level 5 for function
void olTest2() */
{
    unsigned int a = 3;
    unsigned int b = a + 5;
    a = b - 1;
}

void main()
{
    olTest1();
    olTest2();
}
EXERCISE:

Using the code developed in previous lab session, apply different levels of code optimization and observe the difference. Mention which levels of optimization are to be used for the code and why.

OBSERVATIONS:

____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
Lab Session 11

OBJECT

Relocating 8051 programs

THEORY

Frequently, it is necessary to start your 8051 program at an address other than 0000h. This is typically the case when using a ROM monitor that starts at 0000h. To relocate your 8051 programs, you must perform the following three steps.

1. Adjust the address of the reset vector.
2. Adjust the EPROM starting address.
3. Adjust the addresses of the interrupt vectors.

These steps are easy to accomplish and each is discussed in the following sections. The following examples assume that your new starting address is 8000h.

Before performing relocation, follow the steps below to create the project and c source file.

Steps:

- Run Keil μVision.
- Select ‘Project’ menu and ‘New Project’.
- Enter the project name “MyProject”
- Select the target microcontroller.
- Then press ‘OK’
- A pop-up window appears.
  - Press ‘YES’ to include the Standard 8051 Startup Code to your project.
- Once the project created you need to create your source code file. Here in the example we have created a file named cfile.c.
- Add the file to the project.
  - Right click on the ‘Source Group 1’ item of your project.
  - Select ‘Add Files to...’
  - Add your source code file.
- Add some source code to the file to complete the application.

1) RESET VECTOR ADDRESS

Keil C51 comes with source for the startup code. This is where the reset vector is initialized. The startup code can be found in \C51\LIB\STARTUP.A51. You should first make a copy of this file to use with each individual project you create.

The relevant lines of STARTUP.A51 appear in the following figure:
Change the line that reads \texttt{CSEG AT 0} to read \texttt{CSEG AT 8000h}. This tells the assembler that the reset vector is at 8000h.

2) \textbf{EPROM START ADDRESS}

Open the \textit{µVision ‘Options for Target...’} window. Select the ‘Target’ tab. Set your EPROM Start to 0x8000.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig11.1}
\caption{Startup code window}
\end{figure}
3) INTERRUPT VECTOR ADDRESS

Open the µVision ‘Options for Target...’ window. Select the ‘C51’ tab. On the C51 tab, set ‘Interrupt vectors at address’ to 0x8000.

The next time you debug your project, all of your source code will be located at 8000h and above as shown in the following figure:
EXERCISE:

Create a program for adding two numbers and displaying their result. The program should start from the location 4000H. Mention the steps briefly to relocate the program.

OBSERVATIONS:

___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________
Lab Session 12

OBJECT

Introduction to DSP Board and Code Composer Studio 3.1

THEORY

TI C6416 DSK Overview:

The digital signal processor is considered to be the heart of the main system. It is designed to perform the beamforming as well as the source localization task. The processor used is a Texas Instruments C6416 DSP which operates at clock speed of 1 GHz. The processor comes preinstalled onto a DSP Starter Kit by Spectrum Digital, called as the TMS320C6416T DSK board. The DSK board makes it easier to interface the DSP with external peripherals, analogue to digital converters, external memory units, power supplies, and is controlled using the CCS (Code Composer Studio) by Texas Instruments through a USB interface from a PC running on normal Microsoft Windows environment. The CCS Studio software comes with a C compiler and an assembler which, helps in programming and controlling the Digital Signal Processor from the pc over the USB interface. The DSK board also comes with four connections, two for input and two for output audio devices. A microphone input port, a line in port, a line out port, and a headphone port. Each of these ports is interfaced to the DSP C6416 through onboard A/D and D/A converters as well as the preinstalled codec on the DSK board i.e., AIC23. Figure below shows an overview of the Spectrum Digital DSK board and the AIC23 Codec.

Fig: 1.1 Spectrum Digital DSK board with AIC23 Codec

The Line In port is used for sound input from a device that is powered by a preamp. It can be used with any stereo microphone input. For this project a 3.5mm stereo jack is used to connect
the microphones to the board. The AIC23 Codec is used to program the onboard A/D and D/A converters, set the sampling rate and control the input and output data through the four ports; Mic in, Line in, Headphones and Line out. The codec is controlled by a debug software i.e. CCS Studio 3.1 that is provided with most of the TI C6000 DSP boards. CCS Studio can be used to program the serial ports, USB ports, dip switches, LEDs, AIC23 Codec, input and output ports on board the DSK. A basic tutorial on using the CCS Studio 3.1 is provided in further topics below. The figure below shows the AIC Codec with the A/D and D/A converters connected to the input and output ports of the DSK Board.

**CCS Studio 3.1 Tutorial:**

1. **Starting the CCS Studio and DSK Board:**
   Double click the icon on the desktop named 6416 DSK CCS Studio v3.1. The program should open with the window as in figure 2.3. At this moment make sure the DSK Board is connected to the PC via the USB link before CCS Studio 3.1 was opened. (Note: CCS Studio 3.1 is not compatible with windows vista). Now we connect the DSK Board to the CCS Studio software.

2. **Connecting DSK Board to CCS Studio:**
   To connect the DSK Board to the CCS Studio software click on debug and then click on connect in the menu that pops down. The figure 1.4 below illustrates this:
Fig: 1.3 Welcome Window of CCS Studio 3.1

Fig: 1.4 Connecting DSK Board to CCS Studio
Once the DSK Board is successfully connected to the CCS Studio software the red dot in the lower left hand corner of the window should turn green with a “connected” message. The figure 1.5 below illustrates this:

**Fig: 1.5 CCS 3.1 Showing DSK board connection status**

3. Opening a project in CCS Studio 3.1:
Most of the projects for CCS Studio are stored by default in the folder named against project name inside a folder with the address: C:\CCStudio_v3.1\MyProjects\YourProject depending upon which directory the software is installed into. It is always recommended to install the CCS Studio into the primary C: drive. Each project consists of the main C source code and some additional headers and support files included for the DSK Board to load the source code into its flash memory and execute it. For introduction purpose let us focus on Audio Project as a sample project in Audioapp folder. To open and load the project into CCS Studio 3.1, go to Project and click on open, the following window should pop up:

**Fig: 1.6 Project Open Window**

Click on the 6416_DSK_INOUT.pjt and then click Open. The project is loaded and on the left column you should be able to see the project explorer with all the source and header files for this project. Click on the + sign to the side of Source folder to expand it and double click on the
inout_2_channels.c to open the C source code at right side that executes the necessary beamforming functions and includes the code necessary to control the AIC23 Codec to sample the input data at 48kHZ as well as store them into different buffers. This is illustrated in the figure below:

![Fig: 1.7 Project explorer with Code editor window](image1)

4. Generating Executable file for DSK board:
To generate the executable file that runs on the DSK Board, click on Project and Rebuild All. The figure below illustrates this:

![Fig: 1.8 Generating Executable file for DSK board](image2)
Once the exe file is made click on: File and the select Load Program. A new window popup. Go to debug folder and load the file named: C6416_DSK_INOUT. Now the file has been loaded onto the DSK Board flash memory, therefore to execute it just press F5 key. To halt the process press Shift+ F5. This process can also be done from debug menu.

5. Result output display:

To check the output results, the direction of sound source can be seen from the LEDs on the DSK board. As the object is moved around the microphone array, the LEDs change turn on and off. The two LEDs in the middle show the sound source in the front direction, the left most and right most indicate whether the sound source is in to the left or right. To check the output results on the PC double click on the files named: Sender and Decision_emulate in the debug folder. Complete path is: C:\CCStudio_v3.1\MyProjects\audioapp\Debug\ The file named Sender reads the files off a DAT file in the same folder where the CCS Studio dumps the results. It sends it to the Decision_emulate via the network API.