King Fahd University of Petroleum and Minerals
College of Computer Science and Engineering
Computer Engineering Department

COE 400 Digital Systems Design

Lab Manual

Prepared By:
Dr. Abdelhafid Bouhraoua

August 2007

Lab 1: Introduction to the Rabbit core module

Contents

Objectives

The objective of this lab is to provide the students with an introduction to the Rabbit core module.

- The Rabbit System
- The IDE
- The way it works
- First Program (Turning On a LED)
- Lab Work: Writing a program that makes the LED blink.

The Rabbit System

The Rabbit family of core modules provides embedded systems developers with a versatile platform packaged in a series of miniature boards called core modules. These modules are based on several versions of the Rabbit microprocessor (R2000, R3000 and R4000). The module cores used in these labs are based on the R3000 microprocessor.

Figure 1: The Rabbit R3000 microprocessor chip

The Rabbit modules have several features that enable them to be easily integrated into embedded systems solutions.

The Rabbit core module used in this lab is the RCM3600/RCM3610. The RCM3600 is a compact module that incorporates the powerful Rabbit 3000 microprocessor, flash memory, static RAM, and digital I/O ports. The RCM3600 has a Rabbit 3000 microprocessor operating at 22.1 MHz, static RAM, flash memory, two clocks (main oscillator and real-time clock), and the circuitry necessary for reset and management of battery backup of the Rabbit 3000's internal real-time clock and the static RAM. One 40-pin header brings out the Rabbit 3000 I/O bus lines, parallel ports, and serial ports. This core provides the user with the following feature set:

- Small size: 1.23" x 2.11" x 0.62"(31 mm x 54 mm x 16 mm)

- Microprocessor: Rabbit 3000 running at 22.1 MHz

Figure 2: RCM3600 core module

- 33 parallel 5 V tolerant I/O lines: 31 configurable for I/O, 2 fixed outputs
- External reset I/O
- Alternate I/O bus can be configured for 8 data lines and 5 address lines (shared with parallel I/O lines), I/O read/write
- Ten 8-bit timers (six cascadable) and one 10-bit timer with two match registers
- 512K flash memory, 512K SRAM (RCM 3610 has 256K flash memory and 128K SRAM)
- Real-time clock
- Watchdog supervisor
- Connections via header J1 for customer-supplied backup battery
- 10-bit free-running PWM counter and four pulse-width registers
- Two-channel Input Capture can be used to time input signals from various port pins
- Two-channel Quadrature Decoder accepts inputs from external incremental encoder modules
- Four available 3.3 V CMOS-compatible serial ports with a maximum asynchronous baud rate of 2.76 Mbps. Three ports are configurable as a clocked serial port (SPI), and one port is configurable as an HDLC serial port. Shared connections to the Rabbit microprocessor make a second HDLC serial port available at the expense of two of the SPI configurable ports, giving you two HDLC ports and one asynchronous/SPI serial port.
- Supports 1.15 Mbps IrDA transceiver

The programming environment is based on an extended version of the C language. This version is called Dynamic C. A large set of libraries are included within the Dynamic C package. Dynamic C is an integrated development system for writing embedded software. It is designed for use with the family of Rabbit controllers and other controllers from the Zworld family.

The RCM3600 receives its +5 V power from the customer-supplied motherboard on which it is mounted. The RCM3600 can interface with all kinds of CMOS-compatible digital devices through the motherboard.

The Integrated Development Environment (IDE)

Dynamic C is an integrated development system for writing embedded software. It is designed for use with Rabbit controllers and other controllers based on the Rabbit microprocessor.

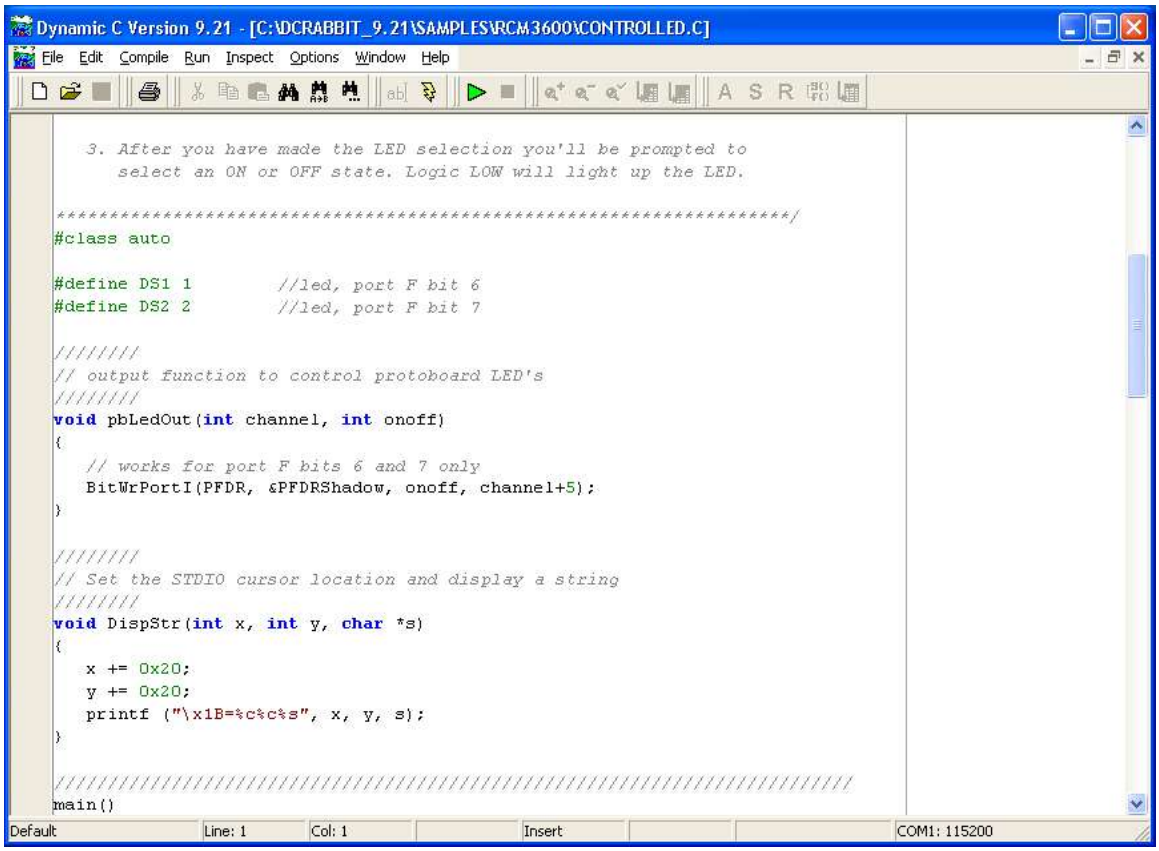

Figure 3: Dynamic C main window

Dynamic C integrates the following development functions:

- Editing
- Compiling
- Linking
- Loading
- Debugging

into one program. In fact, compiling, linking and loading are one function. Dynamic C has an easy-to-use, built-in, full-featured, text editor. Dynamic C programs can be executed and debugged interactively at the source-code or machine-code level. Pull-down menus and keyboard shortcuts for most commands make Dynamic C easy to use.

How does it work?

Designing embedded systems is a very time and resource intensive task. The main steps in realizing the system are often:

- Hardware design: design the hardware that will enable the target application to run.
- Hardware setup: set up all the hardware required for the application to run properly. This includes connecting all the needed connections and power supply wires.

- Design the software algorithm that will be included in the microprocessor core module and that will govern the behavior of the system
- Open the Dynamic C IDE and start writing the code that will implement the desired algorithm
- Connect the programming cable to the host PC
- Run and debug the program

The programming cable, as shown in Figure 4, connects to the host PC (through the PC serial port) on one side and to the RCM3600 on the other side. The edge of the cable on the opposite side of the serial connector contains two connectors. These are 10-pins connectors with 1.27mm spacing organized in two rows of 5 connectors. One of the connectors shows a label on which the word "**PROG**" is written. The other one bears the word "**DIAG"**. Only the connector with the "**PROG**" label is used.

Figure 4: 1.27mm Programming cable

The programming cable is connected on the RCM3600 **J2** connector on the top of the board. Figure 5 shows a zooming of the J2 connector and its location on the board.

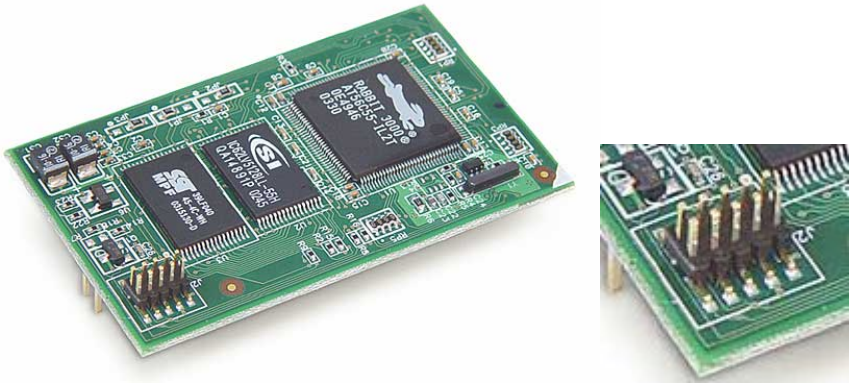

Figure 5: Programming Connector on the RCM3600

The programming cable connects the RCM3600 to the PC running Dynamic C to download programs and to monitor the RCM3600 module during debugging.

Connect the connector of the programming cable labeled **PROG** to header J2 on the RCM3600. Be sure to orient the marked (usually red) edge of the cable towards pin 1 of the connector. Pin 1 is on the side of the connector away from the board's edge.

Running A Sample Program

The following program is a very simple introductory program that simply prints a series of text strings onto the standard output using the C printf statement:

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            Sample_prg.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This is a sample program that displays a group
//                  of text strings on the standard output.
//                  The printf statements used in this program are
//                  executed by the rabbit. The use of printf
//                  statements allows to debug the program while being
//                  executed on the rabbit and not on the PC
//*********************************************************************

#class auto

main() {

int i,j,k;
char prime;

   printf("-------------------------------------------------\n");
      printf(" This is a simple program executed on the rabbit\n");
   printf("-------------------------------------------------\n");

   // print the 1st 100 prime numbers

   printf(" Prime Number = 1\n");
   printf(" Prime Number = 2\n");

   k = 3;
   while(k < 100) {
     i = 2;
     prime = 1;
     while((i<=k/2) && (prime == 1)) {
        j = k/i;
        j = j*i;
        if(j == k) prime = 0;
        else i++;
     }
     if(prime == 1)
```



```
       printf(" Prime Number = %d\n",k);
      k++;
   }
}
```

To get familiar with the rabbit, you are invited to execute the first two programs of the tutorial provided with the documentation.

Lab 2: Timers

Contents

Objective

To learn how to use the internal timers to:

- generate periodic signals
- measure time
- generate internal events

Timers in the Rabbit

There are two timers--Timer A and Timer B. Timer A is intended mainly for generating the clock for various peripherals, baud clock for the serial ports, a periodic clock for clocking Parallel Ports D and E, or for generating periodic interrupts. Timers A1-A7 are general-purpose timers, and Timers A8-A10 are dedicated to specific peripherals. Timer B is more flexible when it can be used because the program can read the time from a continuously running counter and events can be programmed to occur at a specified future time.

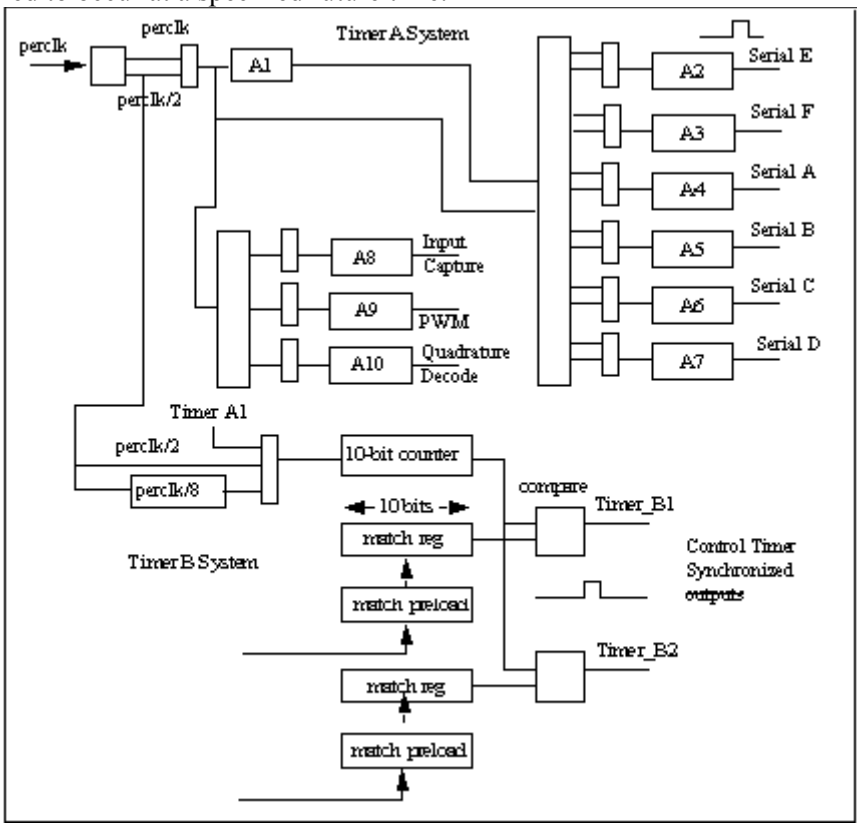

Figure 2.1 – Rabbit Timers

Timer A consists of ten separate countdown timers A1-A10. Timers A1 and A2-A10 are 8-bit countdown registers. The reload register can contain any number in the range from 0 to 255. The

counter divides by (n+1). For example, if the reload register contains 131, then the counter will count for 132 clock pulses. If the reload register contains zero, then the counter realizes a division by one.

The Timer B counter can be driven directly by perclk/2, by that clock divided by 8, or by the output of Timer A1. Timer B has a continuously running 10-bit counter. The counter is compared against two match registers, the B1 match register and the B2 match register. When the counter transitions to a value equal to a match register, an internal pulse with a length of 1 peripheral clock is generated. The match pulse can be used to cause interrupts and/or clock the output registers of Parallel Ports D and E.

For more explanations about timers in the rabbit refer to the R3000 microprocessor documentation.

Timer Experiment

Generate a 50% duty cycle clock of a frequency of 30 kHz. Use the oscilloscope to confirm the period and the duty cycle.

To generate a 50% duty cycle of a frequency of 30 kHz we must do the following:

- Program the timer to produce a frequency equal to the double of 30 kHz which is 60 kHz. The period is 16.66 μs.
- Detect the end of count event from the timer by continuously monitoring the status bit corresponding to the used timer.
- Use the event to toggle a parallel port programmed as output from 1 to 0 and from 0 to 1 everytime the end-of-count event is detected.

The following source code will produce a 50% duty cycle clock of 30 kHz:

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            TIMER_30KHZ.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program uses timers A1 and A2 to produce an
//                  external clock of 30 kHz on a parallel output pin
//                  PB5. It continuously polls Timer A status register
//                  to determine whether the programmed countdown time
//                  has elapsed or not.
//                  When the time is elapsed the clock value is flipped
//                  and set to output PB5.
//*********************************************************************
#class auto

main() {
   int b, toggle;

   toggle = 0;                                  // set the clock to 0
   BitWrPortI(PBDDR, &PBDDRShadow, 1, 5);    // set port PB5 to output
```

```
   //BitWrPortI(PEDDR, &PEDDRShadow, 1, 4);    // set port PE4 to
output

   WrPortI(PECR, &PECRShadow, 0x10);         // transfer high nibble on
                                             //  output of Timer A1

   BitWrPortI(PBDR, &PBDRShadow, 0, 5);      // Initialize PB5 to 0
   //BitWrPortI(PEDR, &PEDRShadow, 0, 4);      // Initialize PE4 to 0
   BitWrPortI(TACSR, &TACSRShadow, 0, 1);    // Disable interrupts
   BitWrPortI(TACR, &TACRShadow, 1, 2);      // Timer 2 clocked by
                                             //  output of A1
   BitWrPortI(TAPR, &TAPRShadow, 1, 2);      // Clock of Timer A1 is
                                             //  perclk/2
   WrPortI(TAT1R, &TAT1RShadow, 0xAF);       // set to 11 to generate a
                                             //  perclk/22 which is
                                             //  1 MHz
   WrPortI(TAT2R, &TAT2RShadow, 0x0F);       // set to 15 to generate a
                                             //  17 us event

   BitWrPortI(TACSR, &TACSRShadow, 1, 0);    // Enable main clock
   while(1) {
     b = 0;
     while (b == 0) {                        // Loop as long as Timer
                                             //  A2 has not reloaded
       b = RdPortI(TACSR);                   // Read the Timer A Status
                                             //  Register
       b = b & 0x04;                         // test bit 2
                                             //  corresponding to
                                             // Timer A2 status
     }
     if(toggle == 0) toggle = 1;             // flip the clock value
                                             // (1 --> 0 and 0 --> 1)
     else toggle = 0;
     BitWrPortI(PBDR, &PBDRShadow, toggle, 5);  // Set clock on PB5
     //BitWrPortI(PEDR, &PEDRShadow, toggle, 4);  // Set next value on
PE4
   }
}
```

Lab Work 1

1. Modify the frequency to 17 kHz.
2. Modify the duty cycle of the clock to 30% and 70%.

Use the oscilloscope to verify the settings.

Lab Work 2

Determine the frequency of the core module using a single timer by measuring the number of times the timer times out during one second. To measure one second, the R3000 has an on-chip real time clock that is accessed through a system variable called SEC_TIMER. This variable is incremented automatically after one second. The variable is of type **`long`**

Lab 3: Parallel Ports Interfacing

Contents

Objective

To enable the students to effectively and efficiently use the parallel ports of microcontrollers.

Introduction to Parallel Ports

A parallel port is a group of I/O that can be controlled (driven) or listened to (read) from inside the microcontroller. These groups of I/O are commonly referred to as "parallel ports" by opposition to "serial ports".
Parallel ports are characterized by the following features:

- Software selectable direction (input or output)
- Software accessible wire values through registers that are directly connected to the IO pins.
  - Ability to read the logic values currently driving the wire connected to the IO pin (in the case it is programmed as input).
  - Ability to drive the wire connected to the IO pin with the desired logic value (in the case it is programmed as output).

Parallel Ports in the Rabbit

The Rabbit has seven 8-bit parallel ports designated A, B, C, D, E, F, and G. The pins used for the parallel ports are also shared with numerous other functions. The important properties of the ports are summarized below.

- Port A--Shared with the slave port data interface and auxiliary I/O data bus.
- Port B--Shared with control lines for slave port, auxiliary I/O address bus, and clock I/O for clocked serial mode option for Serial Ports A and B.
- Port C--Shared with serial port data I/O.
- Port D--4 bits shared with alternate I/O pins for Serial Ports A and B. 4 bits not shared. Port D can be configured as open drain outputs. Port D also contains output preload registers that can be clocked into the output registers under timer control for pulse generation.
- Port E--All bits of Port E can be configured as I/O strobes. 4 bits of port E can be used as external interrupt inputs. One bit of port E is shared with the slave port chip select. Port E has output preload registers that can be clocked into the output registers under timer control for pulse generation.
- Port F-- As outputs, Port F can be configured as open drain outputs. Alternatively, Parallel Port F outputs can carry the four Pulse-Width Modulator outputs. As inputs, Parallel Port F inputs can carry the inputs to the two channels of the quadrature decoders. Port F pins can also be configured to be used as clock pins for clocked Serial Ports C and D.
- Port G--As outputs, Port G can be configured as open drain outputs. Port G inputs and outputs are also used for access to other serial peripherals on the chip such as those used for asynchronous or SDLC/HDLC communication.
- Parallel Ports D-G behave in the same manner when used as digital I/O.

In the RC3600, not all the ports from the R3000 are accessible in the connector. Figure 3 shows the J1 connector that depicts the accessible ports in the RCM3600.

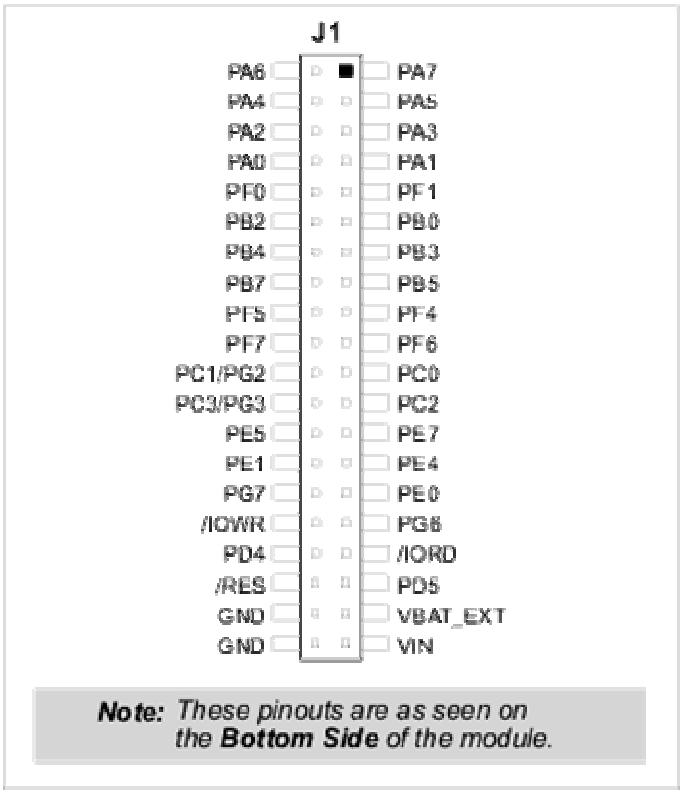

Figure 3.1: RCM3600 Pinout

Toggle switch controlled LED

The first experiment using parallel ports is to connect a toggle switch to a microcontroller and use it to control the lighting of a LED. The setup, as shown in Figure 3.2 below, uses two parallel port pins PB3 and PB5. PB3 is connected to the toggle switch and PB5 is driving the LED.

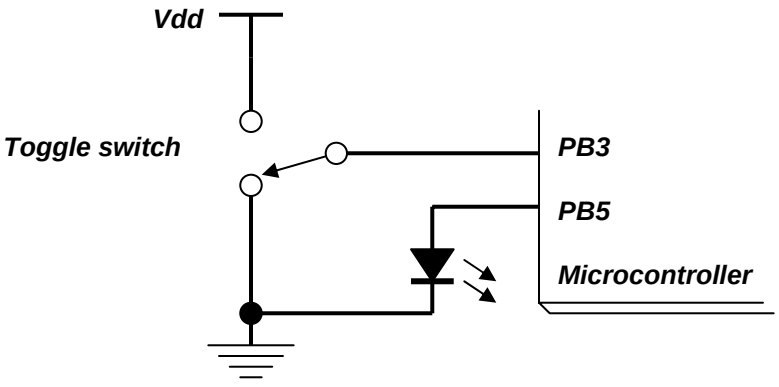

Figure 3.2: Toggle switch controlled LED Experiment setup

The toggle switch is a switch that has 3 connectors. Two connectors are edge connectors and a middle connector. The middle connector is connected to either one of the edge connector alternatively according to the switch contact position.

The following program realizes the control of the lighting of the LED using the toggle switch.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            Sample1.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program changes the status of a LED using the
//                  value input from a toggle switch.
//                  The toggle switch is connected on PB3 and the LED
//                  is connected on PB5.
//                  Change the value only when the toggle switch first
//                  goes to 1. Use the "toggle" flag to ensure that
//                  the LED status is not changed continuously when
//                  the toggle switch is connected to Vdd.
//*********************************************************************
#define TOGGLE_IN 3
#define LED_OUT 5

#class auto

main() {
   int led, toggle;

   led = 0;
   toggle = 0;

   BitWrPortI(PBDDR, &PBDDRShadow, 0, TOGGLE_IN); // Set as input
   BitWrPortI(PBDDR, &PBDDRShadow, 1, LED_OUT);   // Set as output

   BitWrPortI(PBDR, &PBDRShadow, led, LED_OUT);   // The LED is off

   while(1) {
      if((BitRdPortI(PBDR,TOGGLE_IN)== 1) && (toggle == 0)) {
          toggle = 1;
          if(led == 0) led = 1;            // Change the LED value
          else led = 0;
          BitWrPortI(PBDR, &PBDRShadow, led, LED_OUT);
      }
      if(BitRdPortI(PBDR,TOGGLE_IN)== 0)
          toggle = 0;
   }
}
```


Connecting a single pushbutton switch

Single pushbutton switches are used as keys for enabling user interaction with the embedded system. Pushbutton switches, also called momentary switches, are switches that make a contact between two terminals when the pushbutton is pressed. As long as the pushbutton is pressed the contact is realized. When the pushbutton is released, the contact is broken. Figure 3.3 shows an example of a pushbutton switch and its schematic symbol.

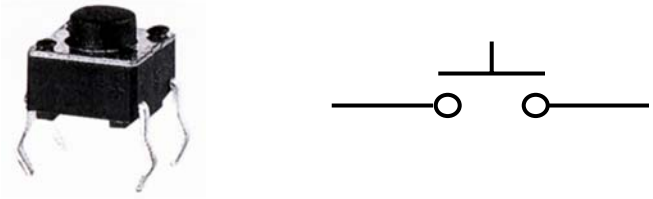

Figure 3.3: Pushbutton switch (example and diagram)

The switch should be connected to one of the parallel ports. The goal is to be able to input an event using the switch. Since a single wire is involved, two states are possible: logic 0 and logic 1. One way is to connect one extremity of the switch to the ground (logic 0) and the other to the parallel port as shown in Figure 3.4.

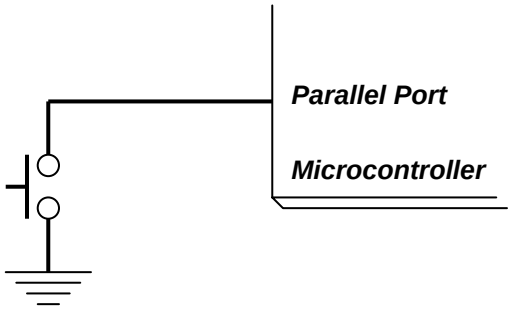

Figure 3.4: Connecting the pushbutton switch to the microcontroller

Figure 3.4 shows that pressing the switch will drive a logic 0 into the parallel port. However, when the pushbutton is released, the value sent to the microcontroller is unknown. What we want is to set the wire to logic 1. A connection from the power supply to the parallel port input will fix that but will create a shortcircuit when the pushbutton will be pressed. To avoid burning our circuit, a resistor should be placed between the power supply and the parallel port input as shown in Figure 3.5.

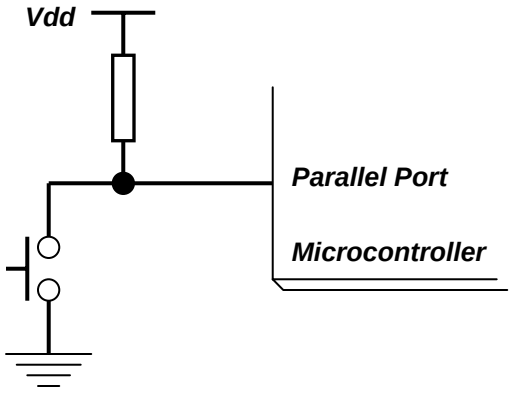

Figure 3.5: Connecting the pushbutton switch to the microcontroller (corrected)

This way, when the pushbutton is pressed the resistor will limit the current flowing from the power supply to the ground. When the pushbutton is released, no current flows through the resistor and the wire is driven up to the power supply voltage.
The value of the resistor should be chosen so that a balance is realized between two conflicting parameters:

- The charge and discharge time of the capacitor that is inherently associated with the wire, which should be fast enough for a proper microcontroller operation. This will require a high current meaning a low resistor value.
- The power consumption that should be kept to a reasonable level. This requires a low current meaning a high resistor value.

Practically, a resistor value between 1K and 100 KΩ is reasonable.

The following program detects a pressed button and prints a message on the standard output.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            pushbutton.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program changes the status of a LED using the
//                  value input from a toggle switch.
//                  The toggle switch is connected on PB3 and the LED
//                  is connected on PB5.
//                  Change the value only when the toggle switch first
//                  goes to 1. Use the "toggle" flag to ensure that
//                  the LED status is not changed continuously when
//                  the toggle switch is connected to Vdd.
//*********************************************************************
#class auto

#define KEY_IN 3

main() {
   int key_flag;

   BitWrPortI(PBDDR, &PBDDRShadow, 0, KEY_IN); // Set as input

   key_flag = 1;

   while(1) {
      if((BitRdPortI(PBDR,KEY_IN)== 0) && (key_flag == 1)) {
          printf("Pushbutton pressed\n");
          key_flag = 0;
      }
      if((BitRdPortI(PBDR,KEY_IN)== 1) && (key_flag == 0)) {
          printf("Pushbutton released\n");
          key_flag = 1;
      }
   }
}
```


It is interesting to run this program and notice what happens. Actually, what happens is everytime the switch is pressed, many copies of the same message are printed. This is due to a mechanical side effect of switches. A switch bounces when pressed and does not settle until some time after it is pressed. This time is small and is in the order of few tens of milliseconds.

Debouncing the switch

Inserting a call to a procedure that introduces a delay of 20 ms fixes the problem. The following program illustrates this change.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            pushbutton_deb.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program changes the status of a LED using the
//                  value input from a toggle switch.
//                  The toggle switch is connected on PB3 and the LED
//                  is connected on PB5.
//                  Change the value only when the toggle switch first
//                  goes to 1. Use the "toggle" flag to ensure that
//                  the LED status is not changed continuously when
//                  the toggle switch is connected to Vdd.
//*********************************************************************
#class auto

#define KEY_IN 3

void wait_20ms();

main() {
   int key_flag;

   BitWrPortI(PBDDR, &PBDDRShadow, 0, KEY_IN); // Set as input

   key_flag = 1;

   while(1) {
      if((BitRdPortI(PBDR,KEY_IN)== 0) && (key_flag == 1)) {
          printf("Pushbutton pressed\n");
          key_flag = 0;
      }
      if((BitRdPortI(PBDR,KEY_IN)== 1) && (key_flag == 0)) {
          printf("Pushbutton released\n");
          key_flag = 1;
      }
   }
}
```



```
void wait_20ms() {
   unsigned long snapshot;

   snapshot = MS_TIMER;          // internal millisecond timer
   while((MS_TIMER – snapshot) < 20);
}
```

Lab Work

Connect and write the code for interfacing a 4x3 keypad switch with the RCM3600. The schematic of the switch and the connector are given in Figures 3.6.

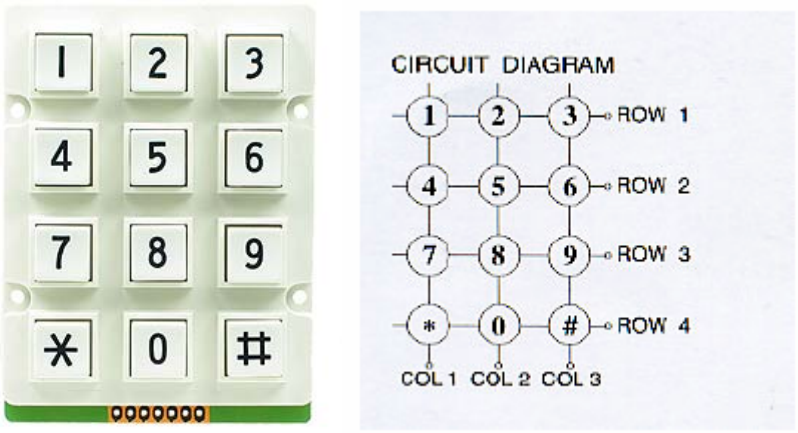

Figure 3.6: Picture and diagram of the 4x3 keypad

The connector is a row connector whose connections are given in the table below.

Output Pin No	Symbol
1	ROW 1
2	ROW 2
3	ROW 3
4	ROW 4
5	COL 1
6	COL 2
7	COL 3

Lab 4: Using the Serial UART

Contents

Objective

To be able to use serial ports

The Rabbit Serial Ports

The Rabbit 3000 has 6 on-chip serial ports designated A, B, C, D, E, and F. All the ports can perform asynchronous serial communications at high baud rates. Ports A-D can operate as clocked ports. Ports A and B can be switched to alternate pins. Ports E and F support SDLC/HDLC synchronous communications in addition to standard asynchronous communications. Port A has the special capability of being used to remote boot the microprocessor via asynchronous, synchronous, or IrDA (asynchronous serial).

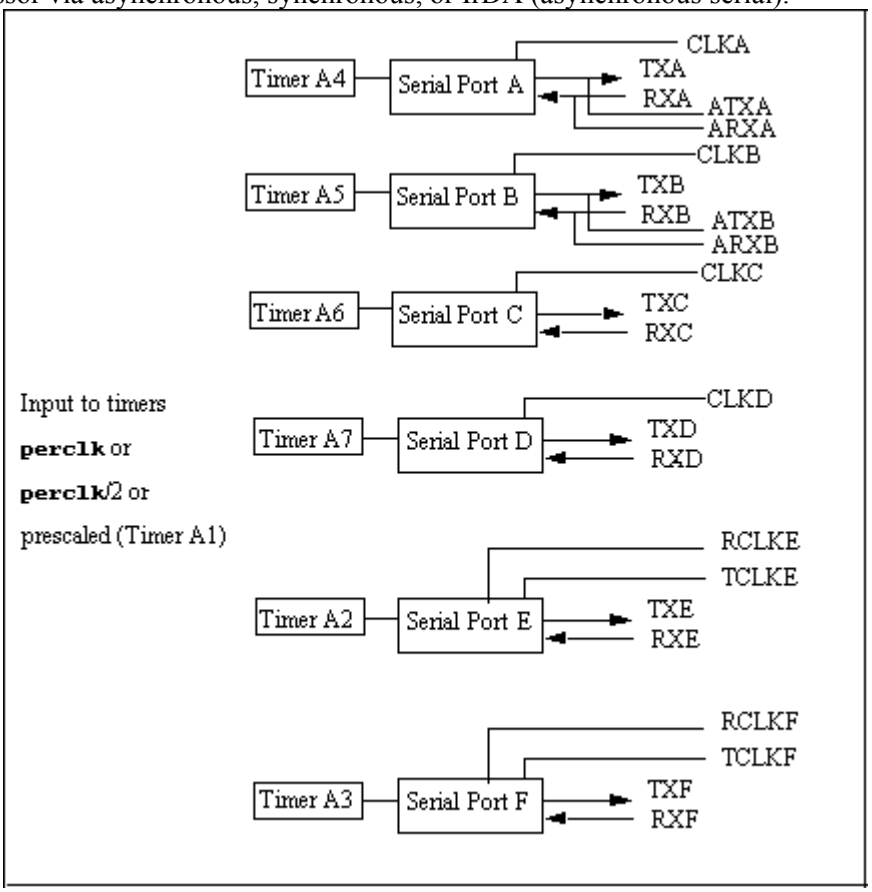

Figure 4.1: Serial Ports

Serial Ports Accessible on the RCM3600 core

There are only five serial ports that are available on the RCM3600. These ports are designated as serial ports A, C, D, E, and F. All five serial ports can operate in an asynchronous mode up to the baud rate of the system clock divided by 8. An asynchronous port can handle 7 or 8 data bits. A 9th bit address scheme, where an additional bit is sent to mark the first byte of a message, is also supported.

Serial Port A is normally used as a programming port, but may be used either as an asynchronous or as a clocked serial port once the RCM3600 has been programmed and is operating in the Run Mode.

Serial Ports C and D can also be operated in the clocked serial mode. In this mode, a clock line synchronously clocks the data in or out. Either of the two communicating devices can supply the clock.

Serial Ports E and F can also be configured as HDLC serial ports. The IrDA protocol is also supported in SDLC format by these two ports.

Either Serial Ports C and D or Serial Port F can be used at one time because these ports share some common pins on header J1. The selection of port(s) depends on your need for two clocked serial ports (Serial Ports C and D) vs. a second HDLC serial port (Serial Port F).

The serial ports used are selected with the **`serXOpen`** function call, where X is the serial port (C, D, or F). Remember that Serial Ports C and D cannot be used if Serial Port F is being used

Sending and Receiving Data Using Two Ports

Two serial ports are used to demonstrate the use of serial ports on the rabbit. These two serial ports are connected as follows (on the breadboard or using wires):

- TXD (PC0) connected to RXC (PC3)
- TXC (PC2) connected to RXD (PC1)

The following program sends a distinct message on every port simultaneously using the **serXputs** function. It receives both messages and stores them onto distinct buffers using the **serXgetc** function.

- The **serXputs** function sends a zero-terminated string over the serial port character by character.
- The **serXgetc** gets a single character from the serial port.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            serial_loop.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program establishes communication between two
```

```
//                  serial ports: port C and port D.
//                  Two distinct text messages are sent from each port
//                  to the other port and are received and stored in a
//                  buffer associated with each port.
//*********************************************************************

#class auto

#define _BD_RATE 19200

#define CINBUFSIZE  31
#define COUTBUFSIZE 31

#define DINBUFSIZE  31
#define DOUTBUFSIZE 31

main() {
   static char msg1[50],msg2[50], buf1[50],buf2[50], rt;
   auto int nIn1, nIn2, cnt1, cnt2;

   rt = 0x0d;
   sprintf(msg1,"This is a message from the C port%c\0",rt);
   sprintf(msg1,"But This message is from the D port%c\0",rt);
   BitWrPortI(PEDR, &PEDRShadow, 0, 5);   //set low to enable serial
device
   BitWrPortI(PCFR, &PCFRShadow, 1, 0);   //set low to enable serial
port
                                     // serial library functions
   serCopen(_BD_RATE);               // open serial session
   serDopen(_BD_RATE);
   serCwrFlush();                    // flush write buffer
   serCrdFlush();                    // flush read buffer
   serDwrFlush();                    // flush write buffer
   serDrdFlush();                    // flush read buffer

   serCputs(msg1);
   serDputs(msg2);
   cnt1=0;
   cnt2=0;
   while((cnt1 != -1) || (cnt2 != -1)) {
      if((cnt1 != -1) && (nIn1=serCgetc() != -1)) {
         if(nIn1 == 0x0d) {
            buf1[cnt1] = 0;
            cnt1 = -1;
         }
         else {
            buf1[cnt1] = (char) nIn1;
            cnt1++;
         }
      }
      if((cnt2 != -1) && (nIn2=serDgetc() != -1)) {
         if(nIn2 == 0x0d) {
            buf2[cnt2] = 0;
            cnt2 = -1;
         }
         else {
            buf2[cnt2] = (char) nIn2;
            cnt2++;
         }
```

```
      }
   }
   printf("Received from C: %s\n",buf1);
   printf("Received from D: %s\n",buf2);
}
```

Lab Work 1

Connect a RCM3600 to the PC and send a predefined text message continuously at a predefined data rate of 19200 bauds.

Lab Work 2

Connect a RCM3600 to a PC serial port and resend text entered from the keyboard to realize the "echo" function.

Lab 5: Signal Acquisition, Generation and Control

Contents

Objectives

DAC Board using the MCP4921

The DAC board available uses a chip from Microchip, the MCP4921 single channel Digital-to-Analog-Converter. The microcontroller side interface of this DAC is serial. It follows the SPI protocol. However, it is unidirectional. Figure 5.1 shows a pinout of the chip.

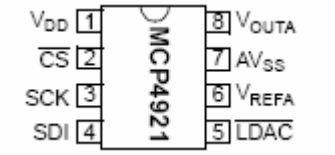

Figure 5.1: MCP4921 DAC

The board is a single channel board that has a 5V reference voltage. Its picture is shown in Figure 5.2. It provides the analog voltage output along with the ground on one edge through a terminal block connector. The digital interface, power supply and ground are accessible through a two-row, 10 pins female header which layout is given in Figure 5.3.

Figure 5.2: DAC board

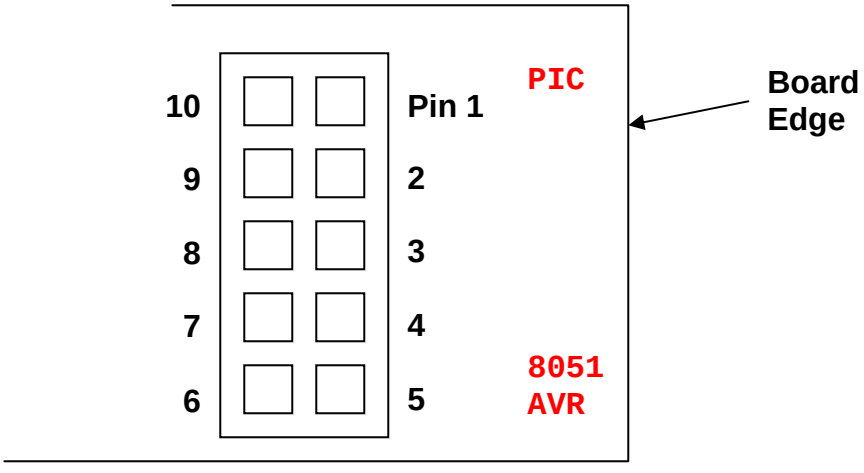

Figure 5.3: Board connector pins

The Board connections are given in the following table:

Board Pin	MCP4921 Signal
1	CS~
2	nc
3	nc
4	nc
5	VDD
6	AVSS & LDAC~
7	Nc
8	SDI
9	SCK
10	nc

The datasheet of the MCP4921 shows the way conversion is performed and how the conversion data is sent to it.

Generating a predefined voltage

The first experiment is to generate a predefined voltage. The voltage of 1.5V is the target voltage. The board schematic shows that the reference voltage is 5V. This corresponds to 4095, the maximum value that can be represented on 12 bits. Therefore, the value corresponding to 1.5V is given by:

Value = (4095 * 1.5) / 5 = 1228.5 approximated to 1229 = 0x04cd in hexadecimal

The following program generates this voltage. Run this program and verify that it actually generates a voltage that is approximately 1.5V.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            DAC_single.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program puts a single voltage on the output of
//                  a small DAC board using the Microchip MCP4921 DAC
//                  chip.
//                  The data is sent to the chip using the SPI
//                  interface. SPI is programmed in assembly for
//                  fast execution.
//*********************************************************************
#class auto

#define DAC_CS 0
#define DAC_SDI 4
#define DAC_SCK 5

static unsigned long flength, block_size, length2;
static unsigned char buf[40];
static unsigned char dt1,dt2;
```

```
void dac_send();
void dac_put_byte();

main() {
   int i;

   // Using bits B0, B3 and B5 as column inputs
   BitWrPortI(PBDDR, &PBDDRShadow, 1, DAC_CS);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, DAC_SDI);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, DAC_SCK);

   BitWrPortI(PBDR, &PBDRShadow, 1, DAC_CS);    // Disable CS

   dac_send(0x04cd);
   while(1);
}

void dac_send(unsigned int smpl) {
   int data, data1, bit, i;
   unsigned char d1;

   data = smpl;
   data = data & 0x0fff;
   printf("data = %04x\n",data);
   data1 = (data >> 8) & 0x00F;
   d1 = (unsigned char) data1;
   dt1 = 0x70 | d1;                // 7 = 0111 = Channel A
                                   //            Buffered
                                   //            1x
                                   //            Not SHDN
   data1 = data & 0x0ff;
   d1 = (unsigned char) data1;
   dt2 = d1;
   printf(" DT1 = %02x\n",dt1);
   printf(" DT2 = %02x\n",dt2);
   dac_put();
}

#define DAC_CS_0 00h
#define DAC_CS_1 01h
#define DAC_SDI_0_SCK_0 00h
#define DAC_SDI_1_SCK_0 10h
#define DAC_SDI_0_SCK_1 20h
#define DAC_SDI_1_SCK_1 30h

#asm
dac_put::
   ld             a, DAC_CS_0
   ioi ld   (PBDR), a
   ld             a, (dt1)
   ld             d, a
   ld             a, (dt2)
   ld             e, a
   ld             b, 16
lp1::
   rl             de
   jr             c, send_one
   ld             a, DAC_SDI_0_SCK_0
   ioi            ld (PBDR), a
```

```
   ld             a, DAC_SDI_0_SCK_1
   ioi            ld (PBDR), a
   ld             a, DAC_SDI_0_SCK_0
   ioi            ld (PBDR), a
   jr             after
send_one::
   ld             a, DAC_SDI_1_SCK_0
   ioi            ld (PBDR), a
   ld             a, DAC_SDI_1_SCK_1
   ioi            ld (PBDR), a
   ld             a, DAC_SDI_1_SCK_0
   ioi            ld (PBDR), a
after::
   dec            b
   jr             nz, lp1
   ld             a, DAC_CS_1
   ioi            ld (PBDR), a
   ret
#endasm
```

Playing back a .wav audio file

Another interesting application is to playback a prerecorded sound file saved using the PC wav format. The wav format description is available at: http://ccrma.stanford.edu/courses/422/projects/WaveFormat/

The following program plays a sound file that is uploaded within the program itself using the **ximport** statement. The output of the DAC is connected to a RCA jack that is introduced in the PC line-in sound input. The PC amplifier will amplify the analog signal that can be heard on the PC speaker. A separate audio amplifier and speaker can be used as well.

The sound file has been created using the Windows sound recorder utility. The sampling rate and format are: PCM 8-bits, 8 kHz, Mono.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            DAC_wav.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program plays back a single audio file on the
//                  output of a small DAC board using the Microchip
//                  MCP4921 DAC chip.
//                  The data is sent to the chip using the SPI
//                  interface. SPI is programmed in assembly for
//                  fast execution.
//*********************************************************************

#class auto

#ximport "C:\Documents and Settings\USER\Desktop\CSource\shaheed.wav"
aya1

#define DAC_CS 0
```

```
#define DAC_SDI 4
#define DAC_SCK 5

static unsigned long flength, block_size, length2;
static unsigned char buf[40];
static unsigned char dt1,dt2;

void dac_start();
int get_data_start();
void dac_send();
void dac_put_byte();

main() {

   // Using bits B0, B3 and B5 as column inputs
   BitWrPortI(PBDDR, &PBDDRShadow, 1, DAC_CS);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, DAC_SDI);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, DAC_SCK);

   BitWrPortI(PBDR, &PBDRShadow, 1, DAC_CS);
   BitWrPortI(TACSR, &TACSRShadow, 1, 0);    // Enable main clock
   BitWrPortI(TACSR, &TACSRShadow, 0, 1);    // Disable interrupts
   BitWrPortI(TACR, &TACRShadow, 1, 2);      // Timer 2 clocked by
output of A1
   BitWrPortI(TAPR, &TAPRShadow, 1, 2);      // Clock of Timer A1 is
perclk/2
   WrPortI(TAT1R, &TAT1RShadow, 0x0A);       // set to 11 to generate a
1 us clk
   WrPortI(TAT2R, &TAT2RShadow, 0x7C);       // set to 124 to generate
a 125 us clk

   dac_start();
}

void dac_start() {
char fnd;
int id;

    xmem2root(&flength, aya1, sizeof(long));
    printf("length = %d\n", flength);
    length2 = 0;
    block_size = 32;
    fnd = 0;
    while(fnd == 0) {
      xmem2root(buf, aya1+4+length2,(int) block_size);
      id = get_data_start();
      if(id != -1) {
         fnd = 1;
         printf("data fnd\n");
         block_size = block_size - id - 4;
      }
      else length2 = length2+block_size;
    }
    while(length2 < flength) {
       printf("length2 = %d\n",length2);
       length2 = length2 + block_size;
       while(block_size > 0){
          dac_send(buf[(int) (32-block_size)]);
          block_size--;
```

```
       }
       block_size = flength - length2;
       if(block_size > 32) block_size = 32;
       xmem2root(buf, aya1+4+length2,(int) block_size);
    }
    printf("DONE\n");
}

int get_data_start() {
    char key[5],wd[5];
    int i,j;

    key[0] = 'd';
    key[1] = 'a';
    key[2] = 't';
    key[3] = 'a';
    key[4] = 0;
    i=0;
    while(i<(block_size-4)) {
      for(j=0;j<4;j++)
        wd[j] = buf[i+j];
      wd[4] = 0;
      printf("wd = %s\n", wd);
      if(strcmp(key,wd)!= 0) i++;
      else return(i+4);
    }
    return(-1);
}

void dac_send(unsigned char smpl) {
   int data, data1;
   unsigned char d1;

   data = smpl;
   data = data - 0x7F;
   data = data * 8;
   data = data + 0x07FF;
   data1 = (data >> 8) & 0x00F;
   d1 = (unsigned char) data1;
   dt1 = 0x70 | d1;
   data1 = data & 0x0ff;
   d1 = (unsigned char) data1;
   dt2 = d1;
   dac_put_byte();
}

#define DAC_CS_0 00h
#define DAC_CS_1 01h
#define DAC_SDI_0_SCK_0 00h
#define DAC_SDI_1_SCK_0 10h
#define DAC_SDI_0_SCK_1 20h
#define DAC_SDI_1_SCK_1 30h

#asm
dac_put_byte::
   ioi ld   a, (TACSR)
   bit            2, a
   jr       z, dac_put_byte
   ld             a, DAC_CS_0
```

```
   ioi ld   (PBDR), a
   ld             a, (dt1)
   ld             d, a
   ld             a, (dt2)
   ld             e, a
   ld             b, 16
lp1::
   rl             de
   jr             c, send_one
   ld             a, DAC_SDI_0_SCK_0
   ioi            ld (PBDR), a
   ld             a, DAC_SDI_0_SCK_1
   ioi            ld (PBDR), a
   ld             a, DAC_SDI_0_SCK_0
   ioi            ld (PBDR), a
   jr       after
send_one::
   ld             a, DAC_SDI_1_SCK_0
   ioi            ld (PBDR), a
   ld             a, DAC_SDI_1_SCK_1
   ioi            ld (PBDR), a
   ld             a, DAC_SDI_1_SCK_0
   ioi            ld (PBDR), a
after::
      dec         b
   jr             nz, lp1
   ld             a, DAC_CS_1
   ioi            ld (PBDR), a
      ret
#endasm
```

Lab work 1

Generate 16 different voltages using the DAC board:

- The voltages are generated one by one in a loop.
- Changing the voltage from the current one to the next is controlled by the user through a pushbutton switch.
- The voltages are chosen so that they cover as much of the 0-5V range as possible.
- Everytime a voltage is generated, the user measures the output of the DAC using a multimeter. The measurements are recorded is the following table:

Index	Hex Value (sent to DAC)	Expected Voltage	Measured Voltage	Error
1				
…				
16				

ADC Board using the MCP3204

The used ADC board is built around the MCP3204 ADC chip from Microchip. The MCP3204 is a four channel ADC that can convert single or differential analog signals. The communication interface with microcontrollers is a 4-wire SPI interface. Figure 5.4 shows a pinout of the chip.

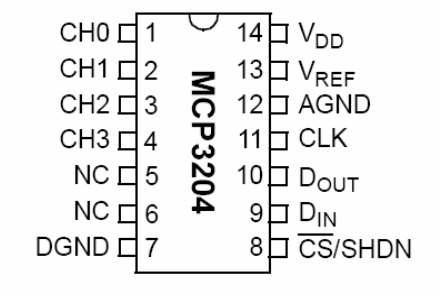

Figure 5.4: MCP3204 ADC

The ADC board is built so that all the four channels are accessible through terminal blocks. The supply voltage is 5V. The reference voltage is, however, 4.09 V. This is a deliberate choice from the designers of the board to make the maximum value that can be represented on 12 bits, 4095, correspond to 4.09 V. This way, a direct correspondence between the voltage value and the produced number is realized without any division. The equation governing the relation between the number produced by the ADC *n* and the input voltage *v* is given below:

$$n = v * 1000$$

Figure 5.5 below shows a picture of the ADC board. The digital interface, power supply and ground are accessible through a two-row, 10 pins female header which layout is given in Figure 5.6.

Figure 5.5: ADC board

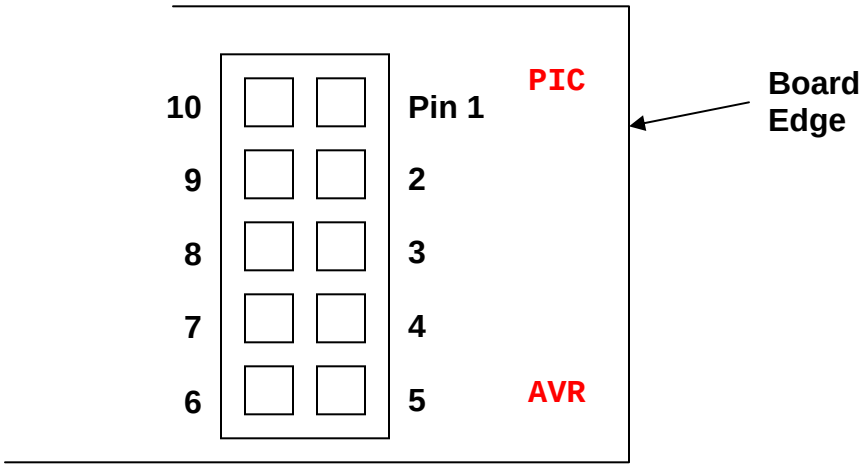

Figure 5.6: ADC board connector pins

Board Pin	MCP3204 Signal
1	CS~/SHDN
2	nc
3	DOUT
4	nc
5	VDD
6	AGND & DGND
7	nc
8	DIN
9	CLK
10	nc

The board has a potentiometer attached to channel 0. The potentiometer input is connected to the reference voltage. The output of the potentiometer is connected to channel 0 input through a jumper.

Sampling and converting the voltage generated on Channel0

The availability of a virtual voltage generation through the potentiometer makes the ADC board directly usable for education purposes without any external component. Thus, the first usage of the ADC board will be to sample and convert a voltage generated through channel 0.
The following program realizes the conversion on channel 0 and displays it on the standard output. It is advisable to vary the potentiometer to change the voltage. The input voltage should be measured with a multimeter to verify that the converted value is reasonably close to the measured one.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            ADC_simple.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program converts a single voltage from channel
//                  0 of a small ADC board using the Microchip MCP3204
//                  ADC chip.
//                  The data is sent to the chip using the SPI
//                  interface.
//*********************************************************************
#class auto

#define DAC_CS 0
#define DAC_SDI 4
#define DAC_SCK 5

#define ADC_CS 7
#define ADC_SDI 4
#define ADC_SDO 2
#define ADC_SCK 5
```

```
#define KEY_IN 0

static unsigned char dt1,dt2;

void wait_10s();
void wait_20ms();
void adc_get(unsigned char channel);

main() {
   int i,b;

   // Using bits B0, B3 and B5 as column inputs
   BitWrPortI(PBDDR, &PBDDRShadow, 1, ADC_CS);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, ADC_SDI);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, ADC_SCK);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, DAC_CS);
   BitWrPortI(PBDDR, &PBDDRShadow, 0, ADC_SDO);

   BitWrPortI(PFDDR, &PFDDRShadow, 0, KEY_IN);

   BitWrPortI(PBDR, &PBDRShadow, 1, DAC_CS);
   BitWrPortI(PBDR, &PBDRShadow, 1, ADC_CS);
   adc_get(0);
   while(1);
}

void wait_10s() {
      unsigned long snapshot;

   snapshot = SEC_TIMER;
   while((SEC_TIMER - snapshot < 2));
}

void wait_20ms() {
      unsigned long snapshot;

   snapshot = MS_TIMER;
   while((MS_TIMER - snapshot < 20));
}


#define ADC_CS_0 0x01
#define ADC_CS_1 0x81
#define ADC_SDI_0_SCK_0 0x01
#define ADC_SDI_1_SCK_0 0x11
#define ADC_SDI_0_SCK_1 0x21
#define ADC_SDI_1_SCK_1 0x31

void adc_get(unsigned char channel) {
   unsigned char d1;
   int i;
   unsigned int data, bit;

   d1 = channel & 0x07;
   d1 = d1 | 0x18;

   //printf("d1 = %02x\n",d1);

   WrPortI(PBDR, NULL, ADC_CS_0);
```

```
   for(i=0;i<5;i++) {
     bit = (d1 >> (4 - i)) & 0x01;
     if(bit == 1){
        //printf("BIT = 1\n");
        WrPortI(PBDR, NULL, ADC_SDI_1_SCK_0);
        WrPortI(PBDR, NULL, ADC_SDI_1_SCK_1);
        WrPortI(PBDR, NULL, ADC_SDI_1_SCK_0);
     }
     else {
        //printf("BIT = 0\n");
        WrPortI(PBDR, NULL, ADC_SDI_0_SCK_0);
        WrPortI(PBDR, NULL, ADC_SDI_0_SCK_1);
        WrPortI(PBDR, NULL, ADC_SDI_0_SCK_0);
     }
   }
   WrPortI(PBDR, NULL, ADC_SDI_0_SCK_1);
   WrPortI(PBDR, NULL, ADC_SDI_0_SCK_1);
   WrPortI(PBDR, NULL, ADC_SDI_0_SCK_0);
   WrPortI(PBDR, NULL, ADC_SDI_0_SCK_0);
   WrPortI(PBDR, NULL, ADC_SDI_0_SCK_1);
   WrPortI(PBDR, NULL, ADC_SDI_0_SCK_1);
   WrPortI(PBDR, NULL, ADC_SDI_0_SCK_0);
   WrPortI(PBDR, NULL, ADC_SDI_0_SCK_0);
   data = 0;
   for(i=0;i<12;i++) {
      bit = BitRdPortI(PBDR, ADC_SDO);
      //printf("Read bit = %d\n", bit);
      data = (data << 1);
      data = data | bit;
      WrPortI(PBDR, NULL, ADC_SDI_0_SCK_1);
      WrPortI(PBDR, NULL, ADC_SDI_0_SCK_1);
      WrPortI(PBDR, NULL, ADC_SDI_0_SCK_0);
      WrPortI(PBDR, NULL, ADC_SDI_0_SCK_0);
   }
   printf("ADC Value = %03x\n",data);
}
```

Lab Work 2

Connect the ADC board's Channel other than 0 to the DAC board's output. The DAC board is setup to operate under the conditions of the experiment of lab work 1. After every press on the pushbutton:

- Convert the DAC generated voltage with the ADC and display it on the stdio.
- Generate the next voltage on the DAC's list
- Wait for the pushbutton to be pressed.

Lab 6: Motor Control

Contents

Objectives

To familiarize the students with the interfacing of motors and their related circuits.

Driving a stepper motor

A stepper motor is an electric motor that can divide a full rotation into a large number of steps, for example, 200 steps. These motors have several coils that are individually controlled. Each coil can be controlled using a single wire. When the proper sequence is applied to the wires controlling the motor, the motor's position can be controlled precisely.

The setup used is an educational kit that is built around a small stepper motor that has four coils. The coils are controlled by 4 inputs labeled INA, INB, INC and IND. Each input is connected to a Darlington transistor that drives one of the coils. The board picture is given in Figure 6.1.

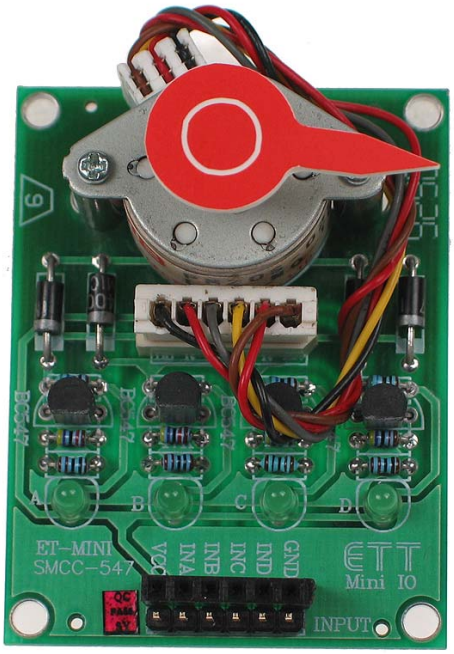

Figure 6.1: Stepper motor board

The control of stepper motors is widely known. The following link gives good explanations on how to control a stepper motor: http://www.doc.ic.ac.uk/~ih/doc/stepper/control2/sequence.html

The principle is to enable one coil at a time in a sequence. The sequence is designed to enable two consecutive rows sequentially so that a full rotation is achieved after all the whole sequence is completed. The following table shows the sequence items. It starts with enabling Coil A while all the other coils are disabled. It continues step by step enabling successively every coil, one by one. To every sequence item corresponds a rotation of the motor by a quarter round.

Sequence	Coil A	Coil B	Coil C	Coil D
1	**on**	off	off	off
2	off	**on**	off	off
3	off	off	**on**	off
4	off	off	off	**on**

Figure 6.2 below shows the application of the sequence

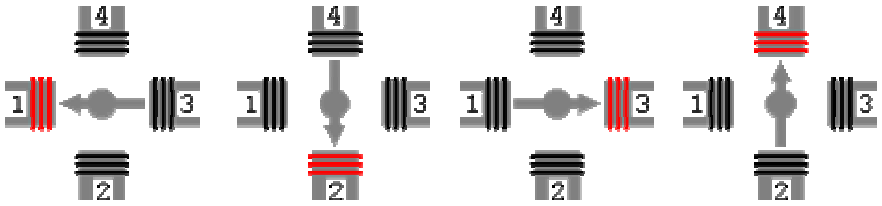

Figure 6.2: Stepper motor sequences

The previous sequence suggests that the angle by which the motor turns is 90 degrees. To reduce the step from 90 to 45 degrees, the following sequence is applied:

Sequence	Coil A	Coil B	Coil C	Coil D
1	**on**	off	off	off
2	**on**	**on**	off	off
3	off	**on**	off	off
4	off	**on**	**on**	off
5	off	off	**on**	off
6	off	off	**on**	**on**
7	off	off	off	**on**
8	**on**	off	off	**on**

The following program generates the sequence items used to turn the stepper motor by 45 degrees. A pushbutton switch is used to make the user enable the next step.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            Stepper.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     The following program generates the sequence
//                  items used to turn the stepper motor by
//                  45 degrees. A pushbutton switch is used to
//                  make the user enable the next step.
//*********************************************************************

#class auto

#define KEY_IN 0
#define INA    3
#define INB       2
```

```
#define INC       4
#define IND       5

void wait_20ms();

main() {
   int key, seq_num;

   BitWrPortI(PBDDR, &PBDDRShadow, 0, KEY_IN);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, INA);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, INB);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, INC);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, IND);

   BitWrPortI(PBDR, &PBDRShadow, 0, INA);
   BitWrPortI(PBDR, &PBDRShadow, 0, INB);
   BitWrPortI(PBDR, &PBDRShadow, 0, INC);
   BitWrPortI(PBDR, &PBDRShadow, 0, IND);

   key = 0;
   seq_num = 0;

   while(1) {
      if((BitRdPortI(PBDR,KEY_IN)== 0) && (key == 0)) {
          key = 1;
          seq_num++;
          if(seq_num == 9) seq_num = 1;
          switch(seq_num) {
          case 1:
             BitWrPortI(PBDR, &PBDRShadow, 0, IND);
             BitWrPortI(PBDR, &PBDRShadow, 1, INA);
             break;
          case 2:
             BitWrPortI(PBDR, &PBDRShadow, 1, INA);
             BitWrPortI(PBDR, &PBDRShadow, 1, INB);
             break;
          case 3:
             BitWrPortI(PBDR, &PBDRShadow, 0, INA);
             BitWrPortI(PBDR, &PBDRShadow, 1, INB);
             break;
          case 4:
             BitWrPortI(PBDR, &PBDRShadow, 1, INB);
             BitWrPortI(PBDR, &PBDRShadow, 1, INC);
             break;
          case 5:
             BitWrPortI(PBDR, &PBDRShadow, 0, INB);
             BitWrPortI(PBDR, &PBDRShadow, 1, INC);
             break;
          case 6:
             BitWrPortI(PBDR, &PBDRShadow, 1, INC);
             BitWrPortI(PBDR, &PBDRShadow, 1, IND);
             break;
          case 7:
             BitWrPortI(PBDR, &PBDRShadow, 0, INC);
             BitWrPortI(PBDR, &PBDRShadow, 1, IND);
             break;
          case 8:
             BitWrPortI(PBDR, &PBDRShadow, 1, IND);
             BitWrPortI(PBDR, &PBDRShadow, 1, INA);
```

```
            break;
         default: break;
            }
         wait_20ms();
      }
      if((BitRdPortI(PBDR,KEY_IN)== 1) && (key == 1))
         key = 0;
   }
}

void wait_20ms() {
   unsigned long snapshot;

   snapshot = MS_TIMER;
   while((MS_TIMER - snapshot) < 20);
}
```

Lab Work 1

The goal is to write a program that will generate the proper sequence to turn the stepper motor with a certain angle (multiple of 45 degrees). It is advised to use the 1/8 round sequence.

Driving a DC motor

DC motors are electric motors that are driven with DC electric current. Their use is widespread in electronically controlled mechanical systems. More precisely, they are used in applications where the precision is way beyond a single rotation like propulsion or translation.

The main issue with DC motors is the speed control. For most applications, the speed is an important factor that will determine system's quality of operation. Another feature in many applications is controlling the rotation direction. DC motors have two electrodes. Connecting the electrodes to respectively power and ground will turn the motor in one direction. Switching the two electrodes will make the motor turn in the opposite direction.

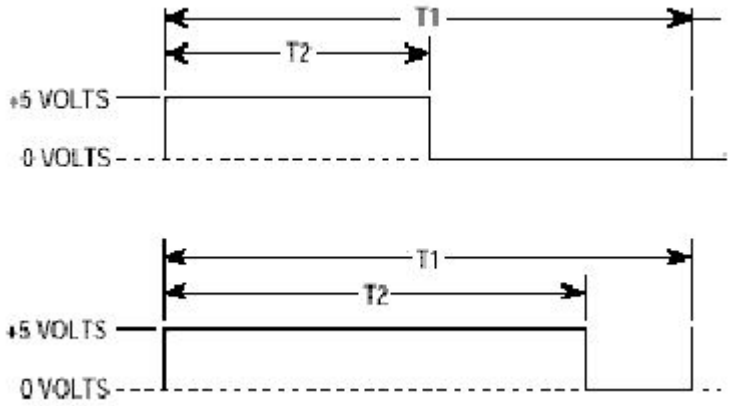

Figure 6.3: Pulse Width Modulation

Controlling the speed of DC motors is realized either through varying the voltage or through delivering the power as pulses. Pulsing on and off the power on one of the electrodes will provide drive to the motor for a fraction of the time. This has an effect to set the speed to a more or less stable value. The pulse frequency and duty cycle will fix the amount of current per unit of time received by the motor coils and thus determines the rotation speed. The speed control logic uses a feedback loop that continuously adjusts the pulse width and frequency. It receives speed

measurement using speed encoders called **optical sensors** or **optical shaft encoders**. These encoders are attached to the motor's shaft. The rotation of a disc that has precisely drilled holes will alternatively cut the light beam of an attached but not rotating optical sensor. The output of this assembly is a square wave digital signal which pulse duration corresponds to the time between two holes. Given the fact that the angle between two consecutive holes is known, it becomes easy to compute the rotation speed of the motor.

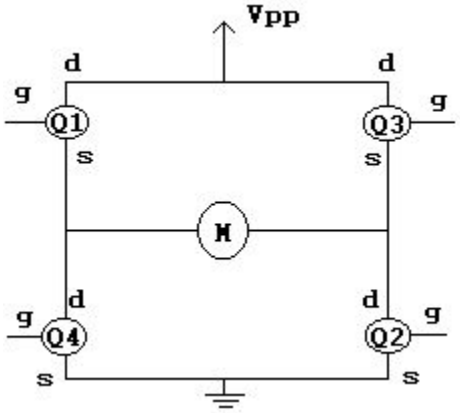

Figure 6.4: Generic H-bridge diagram

Direction is controlled through changing the polarization of the motor. It is obviously impractical to manually change the connections of the motor every time a change of direction is needed. Therefore, this operation is done electronically using power transistors. A structure, commonly called H-bridge, is used for this purpose. Figure 6.4 shows a generic H-bridge.

The used DC Motor board is an educational kit specially designed for learning how to control and monitor DC motors. It is a compact and easy to use mini board that includes a small DC Motor together with a standard L293 control IC (Dual H-bridge) along with a photo-interrupter for measuring motor speed.

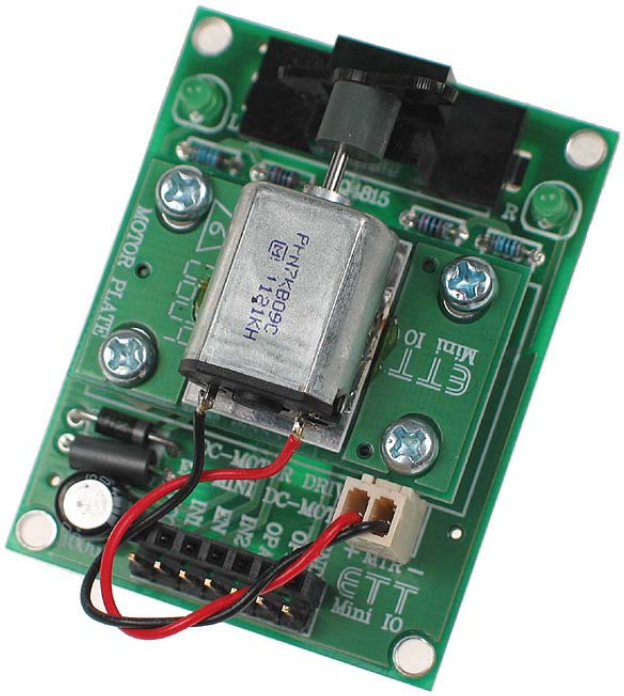

Figure 6.5: DC motor board

Control is easily achieved using a 5 pin control connection. Refer to the L293 datasheet for more information on controlling this unit. Easy to connect standard male and female header pins are included on the board for interfacing to your development board. Together with direction LED's for indication of left or right direction. Figure 6.5 shows the DC motor board. The following table shows how to control the motor through the L293 chip.

EN	IN1	IN2	Effect
0	X	X	Stop
1	1	0	Right
1	0	1	Left
1	1	1	Brake

Changing the direction of the motor

The change of direction in the motor is given by the following program. A pushbutton switch which output is attached to port PB0 (configured as input) is used to enable the change of direction.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            DC_MOTOR_DIR.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program changes the direction of a DC motor by
//                  alternatively driving 10 and 01 on the inputs IN1
//                  and IN2 of the ETT miniature DC motor board.
//                  This board uses the L293 dual H-bridge chip.
//                  A pushbutton switch is used as the user interface.
//                  When pressed, the program changes the direction of
//                  the motor by changing the polarity of the IN1 and
//                  IN2 outputs.
//*********************************************************************

#class auto

#define KEY_IN 0
#define IN1    4
#define ENB       2
#define IN2       3


void wait_20ms();

main() {
   int key, dir;

   BitWrPortI(PBDDR, &PBDDRShadow, 0, KEY_IN);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, IN1);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, ENB);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, IN2);
```



```
   BitWrPortI(PBDR, &PBDRShadow, 0, IN1);
   BitWrPortI(PBDR, &PBDRShadow, 0, ENB);
   BitWrPortI(PBDR, &PBDRShadow, 0, IN2);

   key = 0;
   dir = 0;

   while(1) {
      if((BitRdPortI(PBDR,KEY_IN)== 0) && (key == 0)) {
          key = 1;
          if(dir == 0) {
             BitWrPortI(PBDR, &PBDRShadow, 1, IN1);
             BitWrPortI(PBDR, &PBDRShadow, 1, ENB);
             BitWrPortI(PBDR, &PBDRShadow, 0, IN2);
             dir = 1;
          }
          else {
             BitWrPortI(PBDR, &PBDRShadow, 0, IN1);
             BitWrPortI(PBDR, &PBDRShadow, 1, ENB);
             BitWrPortI(PBDR, &PBDRShadow, 1, IN2);
             dir = 0;
          }

          wait_20ms();
      }
      if((BitRdPortI(PBDR,KEY_IN)== 1) && (key == 1))
          key = 0;
   }
}

void wait_20ms() {
   unsigned long snapshot;

   snapshot = MS_TIMER;
   while((MS_TIMER - snapshot) < 20);
}
```

Setting the speed of a DC motor

The motor speed as mentioned earlier in this document is controlled through Pulse Width Modulation (PWM). The following program illustrates the speed setting by using the millisecond system timer of the Rabbit core MS_TIMER. The PWM is programmed using regular sequence control structures and variables.

```
//*********************************************************************
// King Fahd University of Petroleum and Minerals
// College of Computer Science and Engineering
// Computer Engineering Department
//---------------------------------------------------------------------
// COE-400 Digital Systems Design Lab.
// Author:          Dr. Abdelhafid Bouhraoua
//---------------------------------------------------------------------
// file:            DC_MOTOR.c
//
// Version:         1.0
// Released:        08/20/2007
// Description:     This program sets the speed of a DC motor by
//                  driving a PWM wave on output IN1 of the ETT
//                  miniature DC motor board.
//                  This board uses the L293 dual H-bridge chip.
```

```
//*********************************************************************
#class auto

#define KEY_IN 0
#define IN1    4
#define ENB    2
#define IN2    3
#define OPA    7
#define OPB    5


main() {
   int key, dir, target_speed, pwm_period, pwm_high;
   long pwm_time_snap, pwm_status;
   long opa_time_snap1, opa_time_snap2;

   BitWrPortI(PBDDR, &PBDDRShadow, 0, KEY_IN);
   BitWrPortI(PBDDR, &PBDDRShadow, 0, OPA);
   BitWrPortI(PBDDR, &PBDDRShadow, 0, OPB);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, IN1);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, ENB);
   BitWrPortI(PBDDR, &PBDDRShadow, 1, IN2);

   BitWrPortI(PBDR, &PBDRShadow, 0, IN1);
   BitWrPortI(PBDR, &PBDRShadow, 1, ENB);
   BitWrPortI(PBDR, &PBDRShadow, 0, IN2);

   key = 0;
   dir = 0;
   // set to 120 tpm
   // means 120/60 tps
   // means 120*1000/60 turns per millisecond
   // means 2000 milliseconds is the target period.
   target_speed = 2000;  // in milliseconds
   pwm_period = 50;      // milliseconds
   pwm_high = 10;        // milliseconds

   BitWrPortI(PBDR, &PBDRShadow, 1, IN1);
   pwm_time_snap = MS_TIMER;
   pwm_status = 1;
   while(1) {
      if(pwm_status == 1) {
         if((MS_TIMER - pwm_time_snap) >= pwm_high) {
             BitWrPortI(PBDR, &PBDRShadow, 0, IN1);
             pwm_time_snap = MS_TIMER;
             pwm_status = 0;
         }
      }
      else {
            if((MS_TIMER - pwm_time_snap) >= (pwm_period - pwm_high)) {
             BitWrPortI(PBDR, &PBDRShadow, 1, IN1);
             pwm_time_snap = MS_TIMER;
             pwm_status = 1;
         }
      }
   }
}
```


Lab Work 2

- Write a program that measures the actual speed of the motor by reading output OPA (or OPB) from the DC motor board. The OPA output is set to logic 1 every time the fan blade of the DC motor cuts the photo switch beam. If nothing touches the photo switch, the output is set to logic 0. The DC motor control input IN1 is permanently set to 1 and the speed is measured and displayed every second.
- Write a program that drives the DC motor with the appropriate PWM settings to reach a target speed and monitors this speed by comparing it to the measured speed. The program should change the PWM settings to accommodate the change in speed either by slightly increasing or decreasing the projected speed.

