RED RIVER COLLEGE
OF APPLIED ARTS, SCIENCE AND TECHNOLOGY

STUDENT COURSE PACKAGE

COURSE TITLE: **INTRODUCTION TO MICROCONTROLLERS AND C PROGRAMMING**

COURSE CODE: **PROG2000/PROG2001**

PROGRAM: **ELECTRICAL/ELECTRONIC/INSTRUMENTATION ENGINEERING TECHNOLOGY**

DEPARTMENT: **ELECTRICAL ENGINEERING TECHNOLOGY (EET)**

PRE-REQUISITE: DIGI1030 DIGITAL LOGIC

COURSE DESCRIPTION: This course is designed to provide the knowledge and skills necessary to use a microcomputer for controlling real world industrial applications. A microcomputer development system will be used to control external electronic hardware using the ‘C’ programming language. Simple projects will be used as building blocks for more complex tasks, allowing the student to gradually see the power and potential of a microcomputer and the 'C' language. Hardware interfacing techniques are explained by examples, construction and analysis. Each unit consists of lab exercises, which must be demonstrated, as well as a lab report, which may include answers to assignment questions, C source code and flowcharts.

COURSE FORMAT: 4 hours/week consisting of lecture with a minimum of 2 hours of lab per week. (6 hours/week, for PROG-2001)

COURSE LEARNING OUTCOMES:
After successfully completing this course, students will be able to interface external hardware devices to a microcomputer and write ‘C’ language software to monitor and control those devices by:

- Describing the operation of a microprocessor system
- Writing ‘C’ programs correctly
- Using the Debug tools in the Integrated Development Environment (IDE) to monitor, troubleshoot and repair a ‘C’ program
- Designing and documenting a programming task using pseudo-code and flow charts
- Creating functions that can read or modify entire ports or selected bits of ports
- Interfacing sensors and actuators to input/output ports
- Producing Pulse Width Modulated output waveforms
- Electrically isolating the microcontroller from PWM motor drive electronics

COURSE RESOURCES:

Text

Perry, Greg ***C by Example***, Que ISBN 0-7897-2239-9

Reference

- Zilog Documentation on CD Rom included in the Z8 Encore Microcontroller Development Kit
- Class notes

Equipment

- Oscilloscope (available in the lab)
- L.C.D's, matrix keyboards, electronic components and hook-up wire (provided)
  Students are required to have the following:
- Z8 Encore Microcontroller Development Kit (approximately $50)
- Safety Glasses- **Must be worn when working with your Z8 kit**
- Antistatic Wrist strap- **Must be worn at all times when working with static-sensitive components**
- Breadboard, wire strippers, small needle-nose pliers
- An Electrical/Electronic Department network account

Units of Instruction:

1.0 Microprocessor Concepts

The student will be able to…

1. Describe the internal components of a microprocessor using a block diagram.
2. Describe the bus interface and control signals of a microprocessor from a schematic diagram.
3. Draw the timing diagrams for both RD and WR bus cycles.
4. Describe communication between a microprocessor and a memory device or a port.
5. Compare the functions of the CPU's internal registers.
6. Contrast the function of the address and data busses.
7. Differentiate between Read and Write bus cycles using a timing diagram.

2.0 Getting Started with C and the Z8

The student will be able to…

1. Configure the Z8 IDE *project settings*.
2. Use the Z8 IDE to download and run C programs on the Z8 MCU.
3. Describe the different parts of a C program from a program listing.
4. Describe 'C' data types, variables and constants and the size of each type.
5. Describe a *breakpoint* and its use.
6. *Single step* through a program.
7. Provide a descriptive comment for each line of a C program.
8. Examine the program counter (P.C.) address from the *register* window.
9. Configure the Z8 port registers using C.
10. Identify port pins on the Z8 schematic.
11. Describe the LED interface of the Z8 Microcontroller board, using the schematic.

3.0 Program Flow Control and Bitwise operators

The student will be able to…

1. Distinguish between the correct and incorrect use of C syntax and symbols.
2. Describe arithmetic, relational and logical operators in C programs.
3. Read data from an I/P port using C.
4. Write data to an O/P port using C.
5. Write programs using bitwise “and”, “or”, “xor” and “complement” operators.
6. Describe the term “bit masking” and its purpose.
7. Distinguish between the assignment operator and the comparison operator.
8. Identify and describe the “flow control” statements in a program.
9. Describe the “test” switch interface on the Z8 Microcontroller board using the schematic.

4.0 Arrays and Pointers

The student will be able to…

1. Distinguish between integer and character arrays.
2. Describe how an array of characters differs from a string array.
3. Describe how both an array and a pointer can be used together.
4. Distinguish between an address of a variable and a variable.
5. Use the “address of” and “dereferencing” operators in a program.
6. Use both array subscripts and pointers, to access array elements.
7. Use pointer arithmetic to modify pointer values.
8. Copy the contents of one array to another.
9. Use the “watch” window to examine variables while debugging.
10. Describe a program’s operation using pseudo-code.

5.0 Z8 Hardware interfacing and Delay loops

The student will be able to…

1. Correctly wire an interface to the Z8 Microcontroller evaluation board.
2. Draw a flowchart to describe the operation of a program.
3. Test program conditions using relational and logical operators.
4. Use variables to count program events.
5. Correctly write delay loops using a “for” statement.
6. Distinguish which data type is appropriate to use depending on the variable size.
7. Measure the execution time of a programming event using an oscilloscope.

6.0 C Functions

The student will be able to…

1. Correctly interpret and write function prototypes.
2. Correctly identify parameter data types in function headers.
3. Write functions that accept one or more parameters.
4. Write functions that return a parameter.
5. Correctly make a function call from another function.
6. Write function code based on a supplied flowchart.
7. Debounce mechanical switches using software.

7.0 Pulse Width Modulation

The student will be able to…

1. Describe how a Pulse Width Modulated (PWM) waveform can be produced from a port line on the Z8.
2. Use the C “switch” statement in a C program.
3. Use an array as a lookup table, in a C program.
4. Interface a D.C. motor to a Z8 port line through an electrically isolated drive circuit.
5. Describe the operation of a D.C. motor interface circuit.
6. Write a C program to control the speed of a motor, using PWM.

Safety:

Safety equipment requirements: safety glasses

Students must adhere to all safety protocols. Failure to adhere to all safety protocols, may result in termination from the program.

Attendance:

The student is expected to attend all lectures and labs. If any lectures are missed, the student is still responsible for learning the material. All labs **MUST** be completed to the satisfaction of the instructor before the final exam or a failure will be recorded. Proof of satisfactory completion of all the labs is required prior to validation of the final mark. It is the student’s responsibility to reschedule any missed labs with the Instructor.

Assignments:

All assignments are due 1 week from the date of assignment. Late or otherwise unprofessional submissions will receive a grade of zero.

Labs:

Each lab consists of a demonstration of the exercises for the unit and lab report. The lab manual is a duo-tang folder that is used to hold the report for the lab assignments for each unit. All labs are due within 2 weeks of the assignment date. Labs checked off in this period are worth 5 points. Labs checked off within the 3rd week are worth 3 points. After this, labs must be completed but will receive no additional mark.

Read this first!

This course package includes *unit objectives*, *labs*, *assignments* and additional *reference* information. The reference material in this package is designed to aid the student in understanding details of the hardware and software pertaining to each particular unit of instruction. ***This package is not intended to be a standalone tutorial***.

A separate text on the C programming language is part of the course material and the student is expected to complete the reading assignments as specified in each unit. The information and examples in the text are to be ***supplemented*** by instruction and programming examples provided by the instructor. Likewise, the instructor will make clear the objectives of each unit and illustrate hardware concepts with examples. What you learn in each unit is intended to build on the next.

Doing the assignment worksheets is not optional. They help the student with important concepts, prior to doing the labs. They should be corrected and discussed in class.

Unit 1

UNIT 1: PROG-2000 Introduction to Microcontrollers

Microprocessor Concepts

Unit Objectives:

The student will be able to...

1. Describe the internal components of a microprocessor using a block diagram.
2. Describe the bus interface and control signals of a microprocessor from a schematic diagram.
3. Draw the timing diagrams for both RD and WR bus cycles.
4. Describe communication between a microprocessor and a memory device or a port.
5. Compare the functions of the CPU’s internal registers.
6. Contrast the function of the address and data busses.
7. Differentiate between Read and Write bus cycles using a timing diagram.

In this unit student should do the following:

1. Review as necessary, the following logic devices: “and”, “or”, “xor” logic gates, binary decoders, D-type flip flops, transparent latches and tri-state buffers.
2. Read the provided handouts on Microprocessor concepts (parts 1, 2 and 3).
3. Answer the questions relevant for the unit 1 from the assignments package on a separate piece of paper. Some of these questions will be marked in class.
4. Read chapter #1 of your C text book (“C by Example”). This will give you a brief overview, history and philosophy of the C Language. We will be using many features of the C language, but not all. While we will use a PC to write our programs, the actual code will be downloaded and run (executed) on the Z8 microcontroller. We will be using simple input and output devices on the Z8 to interact with our program. Those examples in the text that refer to input and output on a PC (for example “scanf() from the keyboard and printf() to the display) will not be used.

Unit 1: PROG-2000 Introduction to Microprocessor Hardware, Part 1

Basic Concepts

What is a microprocessor? A microprocessor is a device that is made from digital logic. It that can be programmed with instructions, to perform different tasks. For example, a typical microwave oven is ***controlled*** by a microprocessor. It has a keypad that allows you to select the operation you wish to perform, such as to cook or defrost. It also has a character/digit display that provides you with some visual feedback, for example to tell you how much time is left until your soup is cooked. These are input and output devices which allow you to interact with the ***program*** (the instructions) that is executing on the microprocessor. The keyboard and display are connected to input and output ports on the microprocessor. The program can only perform logical operations on ***data*** already stored in the ***memory*** connected to the microprocessor or from external signals provided by such things as switches or sensors (input devices) connected to an input port of the microprocessor. The program will be written so that it will have rules to follow if some condition occurs. For example, if the microwave is running but someone opens the door, it will shut off the magnatron (the device that generates the microwaves that cook the food) in order to prevent injury to the person operating it. This is because the program is checking the door switch. If the door is ***open*** while the Microwave is ***running*** then it will shut the unit down.

But why use a microprocessor (or a CPU, Central Processing Unit) to do this? Can't I use digital logic to perform the same task? Yes, you can, but as tasks get more complex, it is simpler to write a software program than to produce the logic required to perform a task. We can easily write software to control the microprocessor in the 'C' language, as we will be doing in this course. Programs that are written in 'C' are translated into the native code of the microprocessor by “software development tools” or we could even program it directly in binary machine code.

We will be using a microcontroller development kit called the “Z8 Encore!” (not the Z80) manufactured by Zilog semiconductor. This kit contains a ***Z8 microcomputer***, interface cables, a power supply and software. A ***microcomputer*** consists of a microprocessor, read-only memory (ROM), random access memory or read/write memory (RAM), input/output ports and other peripheral devices all inside the same chip. It really is a computer on a chip. A ***microcontroller*** is just another name for the same thing. The software (Integrated Development Environment) used to program the Z8 in 'C' is loaded on a P.C and the code we develop is sent to the Z8 through a serial cable where is is loaded on the ***flash rom*** of the Z8. The program you wrote will then run on the Z8. Before we start programming let's learn a little bit about hardware that makes up a computer/microcontroller.

A microcomputer has 4 main components:

- microprocessor,
- memory (ROM and RAM),
- input
- output.

The microprocessor communicates with memory or the input/output ports, on a digital multilane pathway called the data bus. On the Z8, the data bus is 8 bits wide labeled D0 (lsb: least significant ***bit***) to D7 (msb). All 8 bits of information are available as a unit (a group). The microprocessor is in control of all data transfers on the Data Bus.

The data bus is a ***bi-directional*** pathway. This pathway consists of 8 wires or copper traces, one for each data line. D7 is on a different wire from D6 and so on. These conductors allow the digital data to pass from outputs (O/P) to inputs (I/P). For example data can be output from the CPU and input to the data lines of a memory (as in a memory write cycle). Or the data can be output from the memory onto the data bus lines and be input to the CPU (a memory read cycle). So “data” (8 binary bits) can use the data bus as its highway in either direction (to or from the MCU), but ***only*** in one direction at a time. This is not unlike a walkie-talkie. If two people are using a walkie-talkie, only one person talks at a time. If they are both talking, then no one is listening. Even worse, bits crash into each other, just as two cars would collide head on if they travel on the same lane, in opposite directions. This error condition, is called ***bus contention***.

Output Ports

Special control signals are generated by the CPU to tell the memory or ports whether the 8 bits of data (one bit on each data line) will travel ***from*** the CPU (a WRITE cycle) or ***to*** the CPU (a READ cycle). These are labeled as the /WR and /RD control signals of the CPU. To better understand this, let’s look at how the CPU can output data to an output port connected to the data bus. We'll use an octal (8) D type flip flop to act as an O/P port. This device has the CLK I/P connected to the clock I/P of all 8 flip flops, internally.

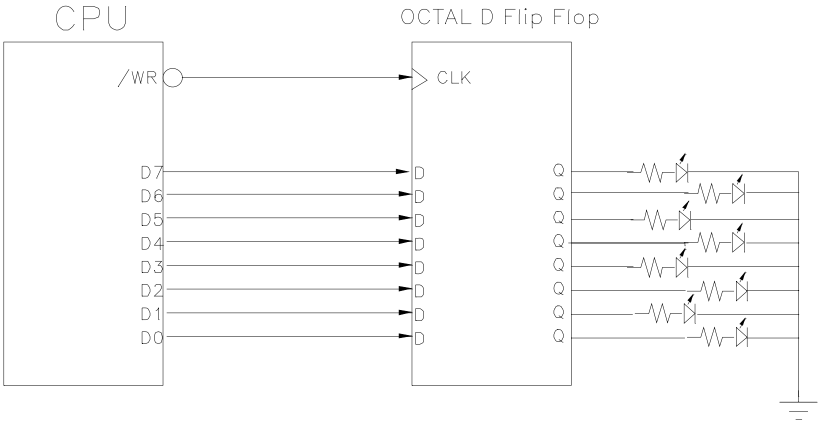


Fig.1

In the above diagram the CPU has placed the following binary data onto the data bus: D7=1, D6=0, D5=0, D4=1, D3=0, D2=1, D1=0, D0=1. (1001 0101)

Since D7 is the msb these 8 binary bits can also be expressed as 95 in Hexadecimal (1001=9 and 0101=5). These 2 hex digits can be used to represent the 8 bits on the data bus. A short time after placing D7-D0 (8 bits) on the data bus, the CPU strobes (or pulses) the /WR control line. What happens? The 8 bits on the data bus are all connected to Data inputs of each D flip flop. When the clock I/P (which is driven from the /WR line from the CPU) goes from low to high, the data on the data bus is clocked into the 'Q' O/P of each flip flop. Fig.2 is a timing diagram which represents what happens. Let’s review the sequence of sending Data from the CPU's data bus to an output port.

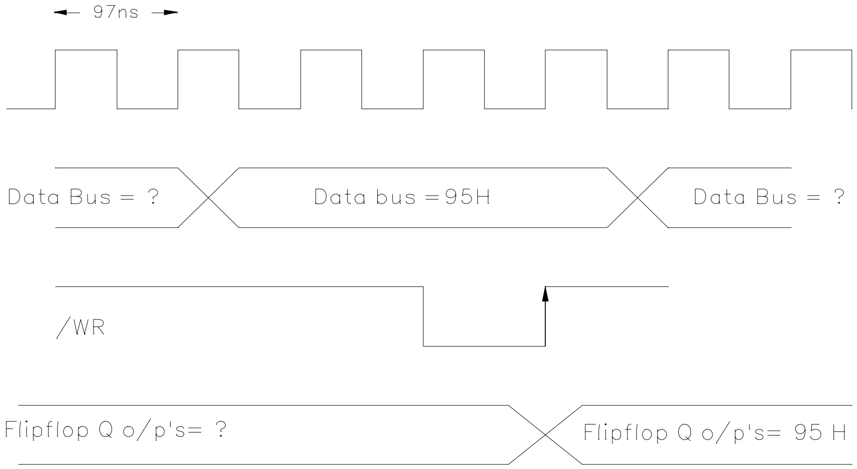


Fig.2

The top row of the timing diagram is the CPU clock. It is called a clock because it really measures time. Its period is 97nsec so it has a frequency of about 10.3 Mhz. Each event in the microprocessors' world occurs for a certain number of clock cycles.

The Data Bus is represented by the overlapping lines on the second row of Fig.2. This is much easier to read than if we put down 8 separate digital lines, set to their logic levels. 95H is the hex value that represents the binary value of the 8 digital data lines D7 to D0 taken as a group (the Data Bus). These lines only contain the stable value of 95H in the time period between the crossover points. The digital values of the data lines of the data bus will contain other values at other times(that's why there is a '?' there).

The /WR control line pulses low for one clock period. As it goes back high it triggers the octal D flip flop to take a ***snapshot*** of the 8 data lines and store them in Q. Since Q is connected to 8 LED's, we can see what value is on the data bus at that particular moment. Notice how the Q outputs change to reflect the contents of the Data Bus on the positive edge of /WR.

Now it's time to review what's been said up to this point. Once you've got it clear in your mind we'll discuss memory, and how it uses the data bus to talk to the CPU.

Memory and the Bi-Directional Data Bus.

Let's look at a single bit of a byte wide memory connected to a CPU.

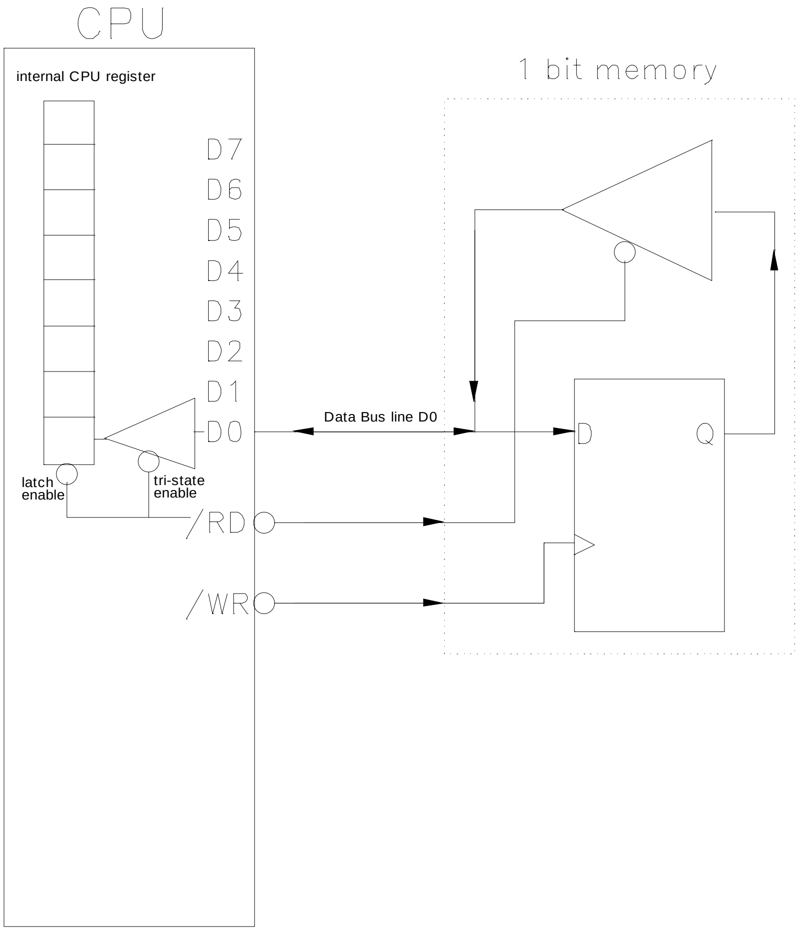


Fig. 3

For clarity, only one ***data line*** (D0) is shown and it is connected to a 1 bit memory. The arrows are shown to illustrate the direction that data travels, depending on which bus cycle is occurring. Even though there are arrows pointing in both directions on D0, remember that data can travel on the data bus, in only ***one direction at a time!***

The WRITE Bus Cycle (/WR)

During a ***/WR bus cycle*** the ***CPU*** outputs a digital signal on D0 and it is available on the D input of the flip flop. The /WR control line becomes ***active*** during this cycle, therefore it is pulses to a logic 0 state. When it returns high, the low to high transition causes the logic level on D0 (hi or low) to be clocked in or stored in the flip flop and available on Q. This is similar to how the data was stored in an O/P port. While this is happening the /RD control line is inactive (because it ***is*** a /WR bus cycle) during the entire cycle. This disables the tri-state buffer, because it requires an active low enable. Therefore this buffer is in the Hi-Z (hi impedance) state and is effectively disconnected from the data bus line D0. That's a good thing. It prevents bus contention (two outputs connected together, fighting to see who can drive the line to its preferred logic level). Remember that during a /WR bus cycle the ***CPU*** is doing the writing to the memory.

The READ Bus Cycle (/RD)

During a ***/RD bus cycle*** the ***memory*** outputs a digital signal on D0 which is available for the CPU to accept. This is how it works. /RD goes low early in the bus cycle and ***enables*** the tri-state buffer of the memory, allowing the data from Q to pass through the buffer to D0. Notice, that the CPU is in control of the operation. The control signals (/RD or /WR) determine which operation will be performed by the memory. Another tri-state buffer is connected internally to a CPU register from the D0 pin of the CPU. Its O/P is connected to the I/P of a latch which is used as a CPU storage register. The internal CPU register can be thought of as an octal D latch (one storage location for each data line).

The next page shows the timing diagrams for both a /WR and a /RD memory cycle.

Fig. 4

/WR Cycle

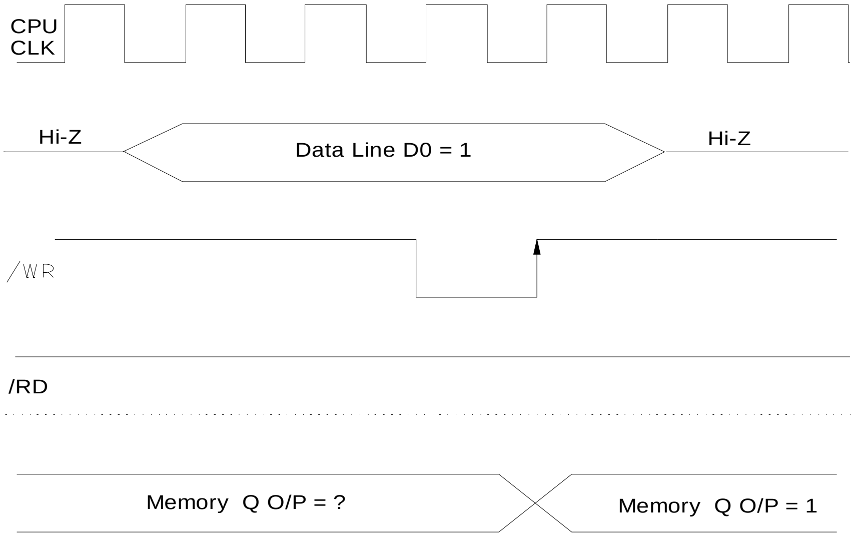
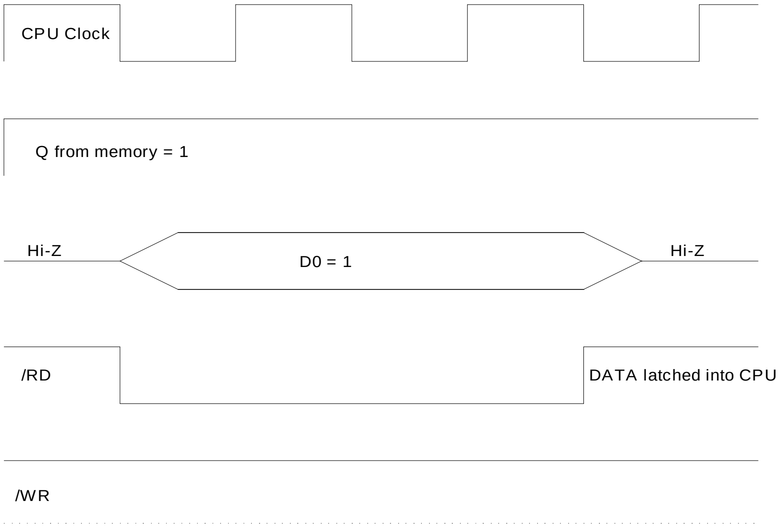


Fig. 5

/RD Cycle



In actual fact all 8 bits are transferred on their respective data lines (D7-D0) at the same time. Instead of one bit of memory, there are eight bits (1 byte) at every memory location. Our next handout will discuss how ***more than one*** memory location can be accessed.

Input Ports

We can use a tri-state buffer as an I/P port. Look at the following schematic:

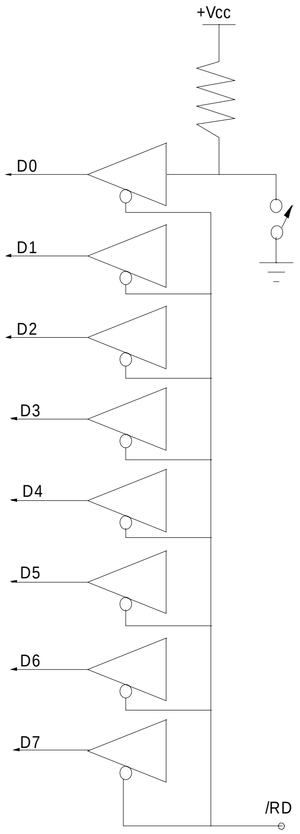


Fig.6

Only one switch is shown for clarity, but each buffer I/P could have a logic switch. When the /RD control line of the CPU becomes active (low in this case) the tri-state buffers are enabled and allow the I/P logic level (depending on the switch position) of the 8 data lines from the buffers to be latched into the CPU. Looks a lot like a memory read cycle, doesn't it? That's because it is!

Unit 1: PROG-2000 Introduction to Microprocessor Hardware, Part 2

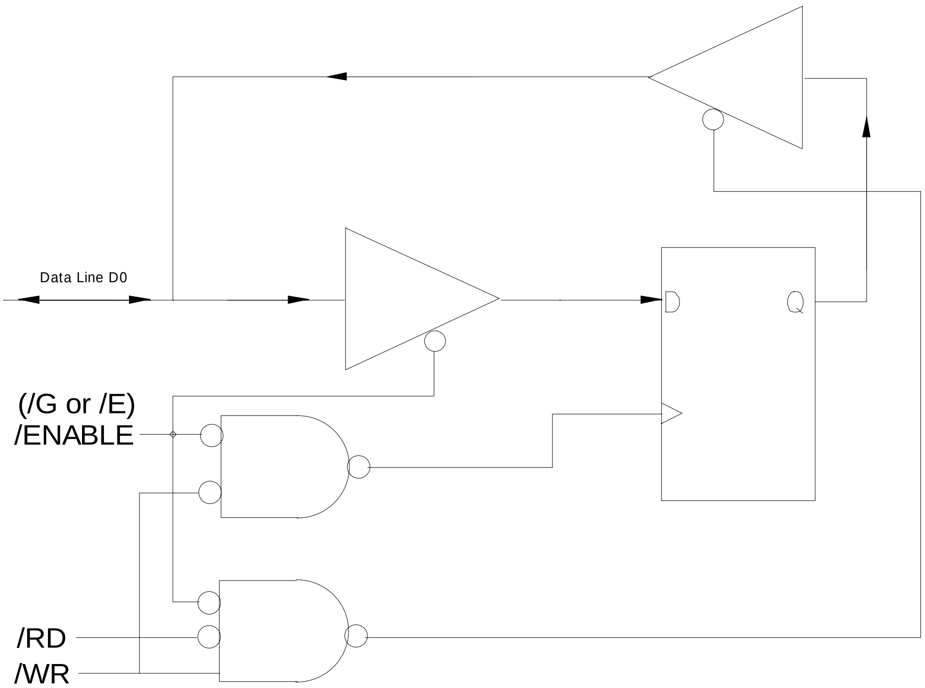
Addressing and the CPU's internal registers

So far we've learned how the CPU transfers ***data*** over the Data Bus, between Memory or Ports. But how does it select ***which port*** or ***which memory location*** it wants to communicate with? The answer is the Address Bus. The Address Bus is a unidirectional bus which is made up of binary ***outputs*** from the CPU. The ***CPU*** is ***always*** the one who decides which memory or port location will be accessed.

How does binary addressing work? Each memory location requires a select line to enable it. If we look at a more realistic depiction of the “1 bit memory” we can see the logic used to accomplish this. Naturally this circuit is duplicated for each bit, D7 to D0.

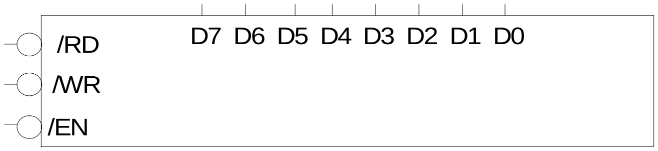
Fig. 1

A more realistic model of a 1 Bit Memory



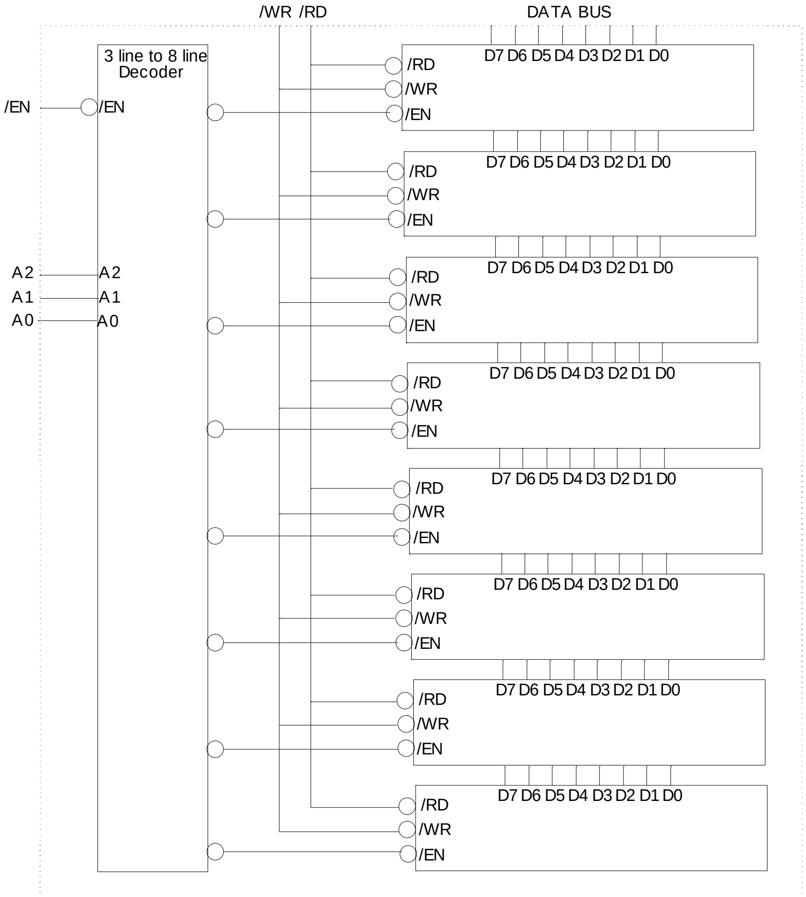
The contents of Fig.1, duplicated for ***all 8 bits*** of a memory location, can be represented by the diagram of Fig. 2. So Fig. 2 is the circuit required ***for each*** memory location. An 8 bit wide memory is required for an 8 bit Data Bus.

Fig. 2



The next diagram, Fig. 3 shows a ***64 bit*** memory organized as 8 locations of 8 bits each (8 rows of 8 bits per row, or 8 bytes of memory). Notice the addition of a 3 line to 8 line decoder which decodes a binary address, that is used to select (enable) a row of memory. There is also a master enable for disabling ***all*** memory locations in the chip. If the memory is disabled, its Data I/P's and O/P's are tri-stated. All memories have a binary decoder as part of their internal circuit. This allows a ***binary address*** to select a memory location. This particular type of memory is referred to as RAM, because ***any*** random location can be accessed. Address lines are always an I/P to RAM (or ROM for that matter).

Fig. 3



64 bit Memory, organized as 8 locations of 8 bits each

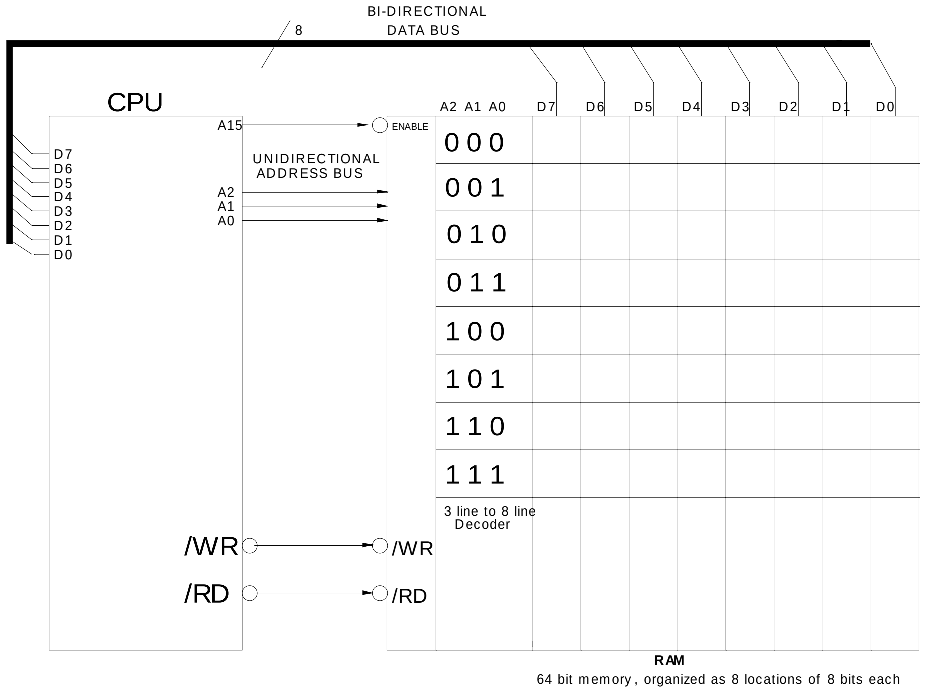
The CPU and the Address Bus

During a READ cycle all 8 bits of a memory or port location are read or input into a CPU data register ***from*** the Data Bus. During a WRITE cycle the CPU outputs 8 bits ***to*** the Data Bus from an internal register. These 8 bits of data can then be latched into a memory or port location.

The Address Bus is always an ***output*** from the CPU. It is used to select a specific memory or port location. A typical 8 bit CPU has a 16 bit Address Bus which means it can select or access 2^{16} or 64K (64 * 1024) memory locations. This is because there are 64K combinations of binary values for 16 bits. Each distinct combination is a unique address. These 16 lines are all part of one Bus so the binary information they contain is taken as a group. The CPU outputs all 16 bits at the same time on the 16 Address lines A15 (MSB) to A0 (LSB).

Its a good thing that addresses from the CPU are provided as binary values. Take a look at Fig. 4 and we'll see why.

Fig. 4



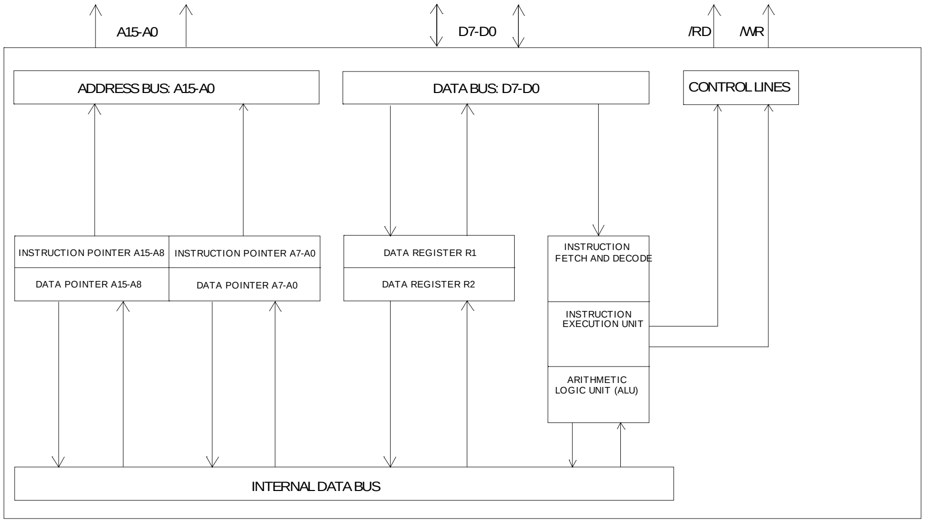
In this memory example we only need to provide 3 wires from A2, A1 and A0 of the CPU to the corresponding I/P's of the memory, to select 8 locations. If the memory did not an have internal binary decoder we would have to run 8 wires instead of 3 (a separate wire to the enable of each row of memory. See Fig. 3). The advantage is more pronounced as our memory size increases. Without binary addressing, a 1K memory would need 1024 wires from the CPU to the memory in order to select each distinct location or row.

With binary addressing we only need 10 lines from the Address Bus (2^{10}=1024) to select every location. It's a lot better to put the binary decode circuitry inside the memory IC and run fewer wires between the CPU and memory. Note as well in this example, that A15 will only allow this memory chip to be enabled if it is a logic 0. Unused address lines can be used for providing select lines or "chip enables" (CE) as A15 is in this case. In other situations A14 and A15 could be used together and connected to an external 2 line to 4 line decoder. This would allow 4 equal sized memory chips to be selected (only one at a time) depending on the state of those upper 2 address lines.

Ports need to have unique addresses, just like memories, in order to prevent bus contention. Address decoders are usually ***not*** part of a Port and so external logic would have to be used to enable it at only one address location. Fortunately for us the Z8 microcontroller has all of its ports and memory already connected and decoded internally, for us. There will still be a few things to learn in order to us to make use of the ports on the Z8, but that is something we will do later.

Fig. 5 shows the internal diagram of an imaginary CPU. Don't worry, it's close enough to the real thing for our purposes. This CPU has 2 internal Address registers that are capable of putting their contents on the Address Bus. Naturally only ***one*** of the two can put its contents on the bus, on any particular bus cycle. During both READ and WRITE bus cycles the address is the ***first*** information made available from the CPU.

Fig. 5



BUS Timing

So, how does the Address Bus fit into the READ and WRITE Bus cycles? Note the timing diagram of a /WR bus cycle in Fig. 6.

- The address bus selects a ***location*** for the data (1234H in this case)
- the data bus supplies the 8 bits of information to the memory location (95H)
- the /WR control line becomes active
- on the low to hi transition of /WR, the contents of the data bus are latched into the memory, completing the cycle.
- The data bus will go hi-Z until the next cycle and the address bus will get a new binary address for the start of the next cycle. The control lines will remain inactive until the next cycle.

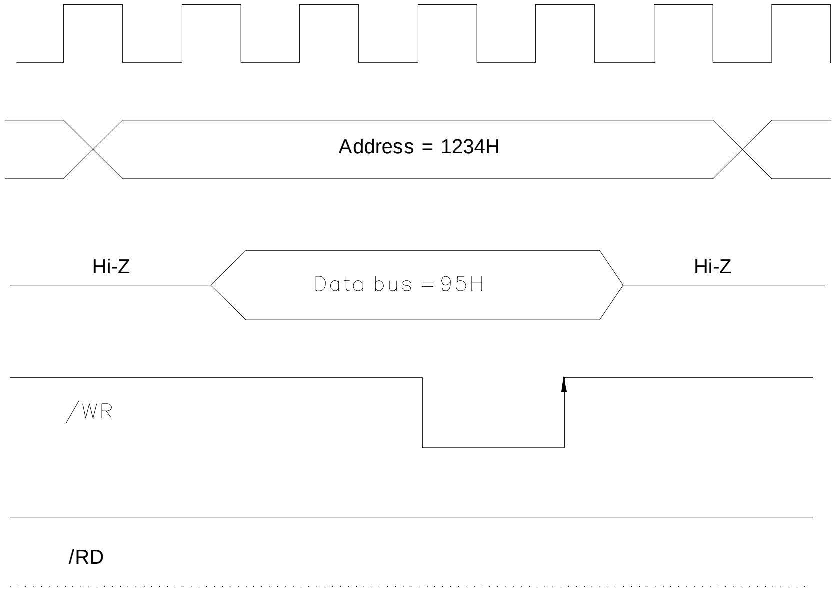


Fig. 6 **WRITE BUS CYCLE**

The timing for a /RD bus cycle is very similar.

Addressing Memory

It's very important to know the difference between the ***address of data*** and the ***data itself.*** Here's one way of looking at it. A mail box at a house address can be like a memory location. When we address an envelope we're saying we want our letter (the data) within the envelope to be delivered to the mailbox at that address. The CPU does something similar. We can specify which locations we want to send or receive data from, by loading different addresses into the address register (like addressing an envelope). The ***data*** will be sent to that location's mail box.

You could also read two memory locations and put the contents of these locations into an address register. In this case the data at these memory locations actually contain an ***address***. However it can't be used as an address unless it is put in a CPU address register. The contents of the address register can then be used to point to some other data at that new memory location. We'll see an example of that later, when we look at the RESET VECTOR. ***Don't worry!*** It's not difficult to understand, once you look at the diagram.

CPU Registers

Let's go back to the internal registers of the CPU as shown in Fig. 5. There you see two ***data r***egisters that have a connection to the Data bus (R1 and R2). They can be the source or destination of what the CPU sends or receives on the data bus. Notice that the CPU also has an ***internal*** data bus. This allows the contents of these registers to be transferred to or loaded from the hi (MSB= Most Significant ***Byte***) or lo (LSB) bytes of the ***address*** registers. As well, if we what to perform some arithmetic or logical operation on data in the registers it can be sent to the ALU and the result sent back over this internal data path.

The data registers can be thought of as temporary storage, like a telephone number on a scrap of paper. Eventually you will want to transfer the data to memory if you want to keep it (the same as putting the number into your telephone book).

The Instruction Pointer (IP)

Notice that one of the address registers is called the Instruction Pointer (***IP***). It is used to point to memory locations that contain ***instructions*** (or code). Whenever we access a memory location addressed by IP, we know that the data from that location is an instruction. We are ***fetching*** an instruction. The box in Fig. 5 labeled “Instruction Decode” and “Instruction Execute” will then figure out which instruction the binary code represents and then perform the operational steps for that instruction. All these operations are what make up an ***Instruction Cycle***. For example let's examine an instruction to perform a logical AND operation between R1and R2 (data registers) and put the result in R1. First we fetch the instruction. Secondly, the contents of both registers would have to be transferred to the ALU (2 steps). Next the logical AND operation would have to be performed. Finally the result is sent back to R1. This instruction took 5 steps!

This is what basically happens during an Instruction Cycle:

- The IP supplies the address of the location of the instruction and the instruction is read on the data bus by the CPU (instruction fetch phase)
- The instruction is interpreted or decoded
- The instruction steps are then executed

Every instruction performs the same steps. After each Instruction Cycle is executed the IP is incremented to the address of the next instruction and the cycle repeats. That is why another name for “IP” is the ***Program Counter*** or PC, because it counts instructions. Some instructions are 1 byte long but others are 2 or 3 bytes in length. Each instruction uses at least one RD bus cycle (one fetch, for each each instruction byte) and the instruction execution itself may perform several additional RD or WR bus cycles, for data transfers. The other address register can be used to point to any other data as needed, and would be used for these additional bus cycles.

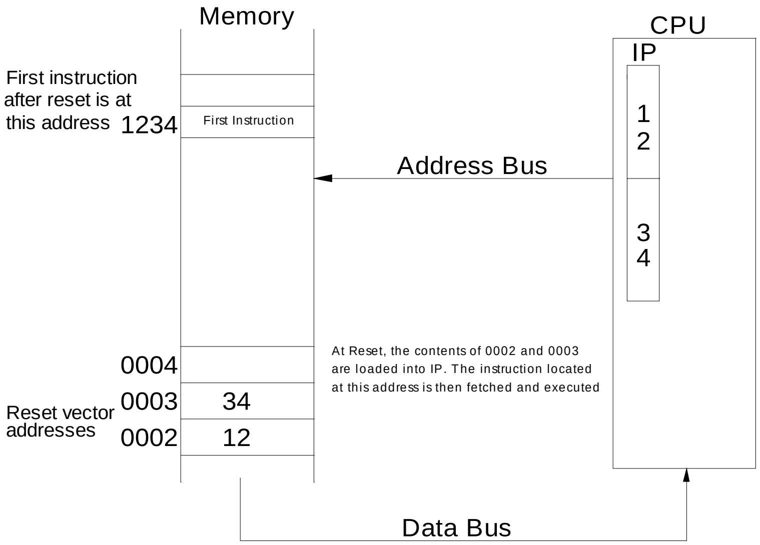
The Z8 has its own instruction set or binary codes that are recognized as instructions. When it accesses data using the IP it ***expects*** the data to be a real instruction, not a number or a character. If the data at this location is not an instruction, the Z8 will likely crash or do something unexpected (just plain wrong).

You can program the Z8 in it's own “machine code” but we will be using the C language instead. C is ideal for programming microprocessors, as you will see. The C compiler and linker will convert (or translate) the C code into instructions that the Z8 can understand.

Resetting the CPU

The CPU is initialized when the CPU is powered up or the RESET input is activated. When this occurs the Z8 “reads” 2 locations in memory called the RESET VECTOR (actually addresses 0002H and 0003H). It then puts the ***contents*** of 0002H in IP MSB (bits 15 to 8) and the contents of address 0003 into IP LSB (bits 7 to 0). Next, it fetches it's first instruction from the contents of the memory address pointed to by IP. The fetch/execute cycle continues on its own after that (see Fig. 7). The instruction must also be located in non-volatile memory (ROM) so it won't be erased when the power is turned off. Personal computers have ROM (the BIOS ROM) for the same reason, to have instructions to start up the computer after it has been off. The Z8 has 64K of Flash ROM for storing instructions (quite a bit for an MCU). The Z8 Integrated Development Environment (IDE) automatically sets things up so that your programs will run when you reset your board.

Fig. 7



At Reset, the contents of 0002 and 0003 are loaded into IP. The instruction located at this address is then fetched and executed

Unit 1: PROG-2000 Introduction to Microprocessor Hardware, Part 3

How to configure the ports on the Z8 Encore!

In order to configure the I/O ports on the Z8 you need to understand the method used to initialize them. The initialization determines how the port will function. In this course we will be using the ports ***only*** for basic I/P and O/P functions or GPIO (General Purpose I/O). Some port pins can be used for an Alternate Function (ALT_FUN) as well as GPIO. Timers, UARTs or A/D converters are examples of port alternate functions. We will disable the use of the alternate function of any port that we use. We must configure any port lines we use, to be either I/P or O/P. In addition O/P ports will be configured for push pull operation (rather than open drain).

The Z8 has eight 8 bit ports labeled A to G and one 4 bit port, H, that are available for GPIO. There are ***4 address locations for selecting the four registers of each Z8 port***. We will use Port A as an example but all ports work the same (ports B, C, etc.). Look at **fig.1** to see how the 4 registers are interfaced to the Z8 CPU. The Z8 address bus is decoded by the address decoder logic. The decoded address O/P's are gated with either RD or WR control lines and used to select the appropriate port register. PAADDR, PACTL, PAIN and PAOUT are each selected by one address. PAADDR is used to select one of five sub registers and PACTL is used to transfer data to the selected sub register. PAIN and PAOUT allow you to ***read*** or ***write*** to the actual port pins. Notice that you can only ***read*** from one of the four port registers, PAIN.

Next, take a look at **fig.2**, titled, “Simplified Port Control Schematic for the Z8”. The schematic is referring ***only*** to port A, but all the ports are similar. Each port line (remember there are 8 bits in each port) can be configured for either I/P or O/P. The schematic shows ONLY the Data Direction (DATA_DIR) and the O/P Control (OUT_CTL) sub registers (see **fig. 3** for a list of all 5 sub registers). As well, ***only bit 2 is shown*** on the schematic, but duplicate circuits exist for ***each bit*** of the port. Analyze the logic. We will be asking questions later.

There are two registers that work together to access and modify the sub registers, PAADDR and PACTL. PAADDR is the Port A Address Register. The Label PAADDR that is I/P to the AND gate is actually a ***decoded address***. Only when ***that*** address is on the address bus AND the ***WR*** signal is active is the PAADDR register enabled to accept the contents of the DATA Bus (the data is then latched into the PAADDR register). In other words we ***Write*** data to the Address register for Port A (PAADDR). This is a Port write operation. The value of that data stored in PAADDR is applied to the I/P's of a 3 to 8 binary decoder. If binary 01H is the data in PAADDR, the decoder’s Y1 O/P becomes active and enables the Data Direction sub register latch. But where does the ***data*** for the “data direction sub register” come from?

It comes from the Port A Control Register (PACTL). This register must be loaded by a second port write, but this time the Port A Control register (PACTL) will be addressed. The byte we send to the PACTL register (on the DATA bus) will be latched into the sub register ***selected*** by PAADDR. Let's say we now write **0x00** to PACTL. Since PAADDR already ***selects*** the Data Direction sub register, all 8 bits in that sub register will be set to 0, even bit 2. This causes the gates that drive the output transistors to be enabled for O/P mode.

Let's look at how this works in more detail. “Data Direction” determines whether a port bit is to be used for I/P or O/P. Each bit in the sub register affects only 1 bit of the port. Some bits of the same port may be I/P while others may be O/P. So, for example if we write 0x0F to the Control Register (0's to bits 7 to 4 and 1's to bits 3 to 0), the 4 msb's bits in that port will be O/P's, while the 4 lsb's will be I/P's. Follow the logic through the gates to the O/P pin. Remember, a ‘1’ on an “OR” I/P will disable the OR gate. The other I/P’s will not be able to change the gate’s state.

“O/P Control” can enable or disable the top transistor in the stacked pair of transistors driving each O/P pin of the port. If it is enabled (by putting a 0 in this bit of the O/P control register), the O/P pin will be able to source or sink current (push pull). If it is disabled (bit is set to a 1), you will need to use a pull up resistor from the Drain of the lower MOSFET, to VDD. We won't be using the open drain configuration in our labs.

In summary, this is how we set all the bits of port A to output mode:

1. Write a value (from data bus) of 01H to the PAADDR register to select the “Data Direction” sub register (Y1 becomes active and enables DATA_DIR).
2. Write a value of 00H to PACTL to latch this value in the selected Data Direction sub register. A '0' at any bit position makes that port line an O/P. A '1' would make the bit an I/P. 0x00 would make all 8 bits O/P.
3. Write a value of 03H to PAADDR to select the “O/P control” sub register.
4. Write a value of 00H to PCTL to set each of the 8 bits of port A to push pull.

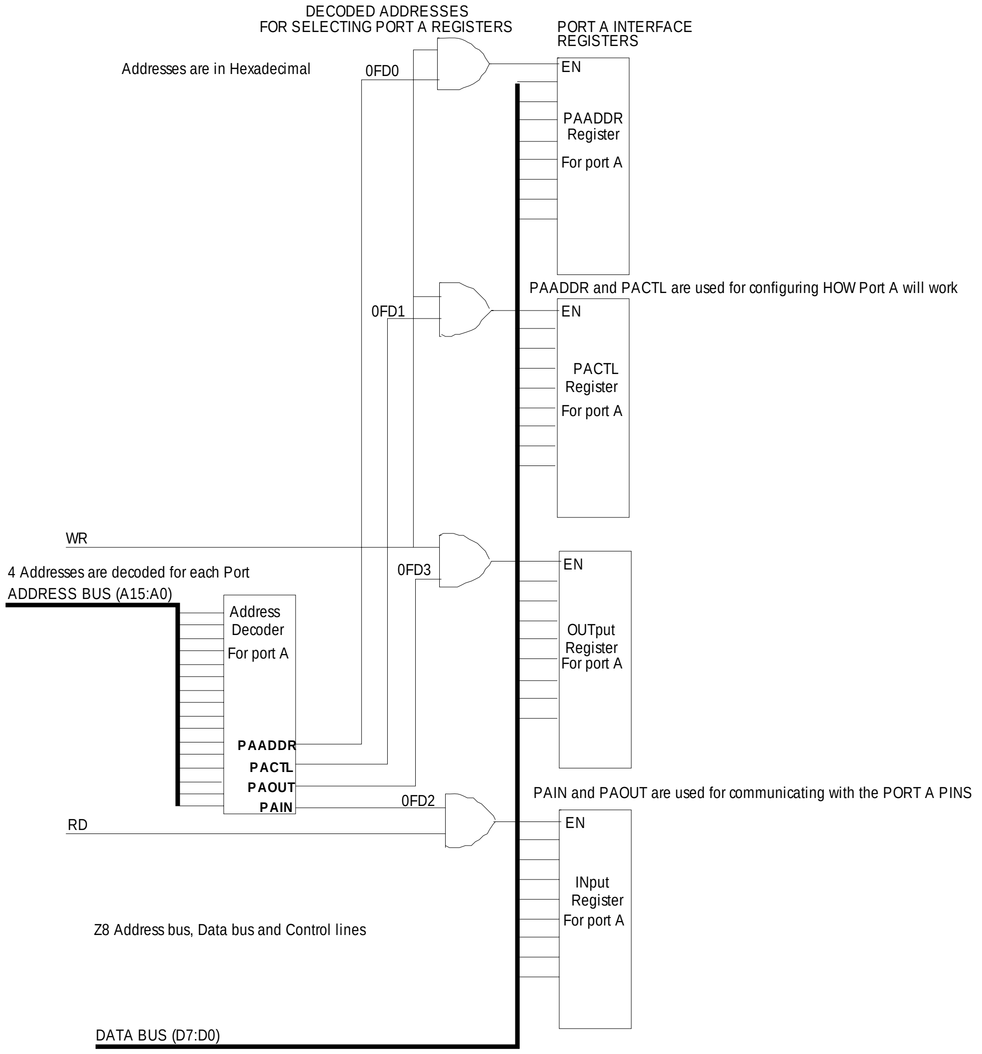
It's a two step process for every function we want to change. First write the sub register address to PxADDR, then we write the sub register data to PxCTL (the x in the labels can be replaced by whatever port you are using ie. A, B, C, etc.).

Once the ports are initialized, how do you read from a port, or write to a port? Simple. You just ***read*** from PxIN to I/P data from the port pins and ***write*** to PxOUT to O/P data to the port pins. ***Remember*** though, that the port you read from must have first been configured as an I/P port, and the port that you write to, must first be configured as an O/P port.

Fig. 1

Registers and Addresses associated with each Z8 Port

Only PORT A is shown.



Addresses are in Hexadecimal

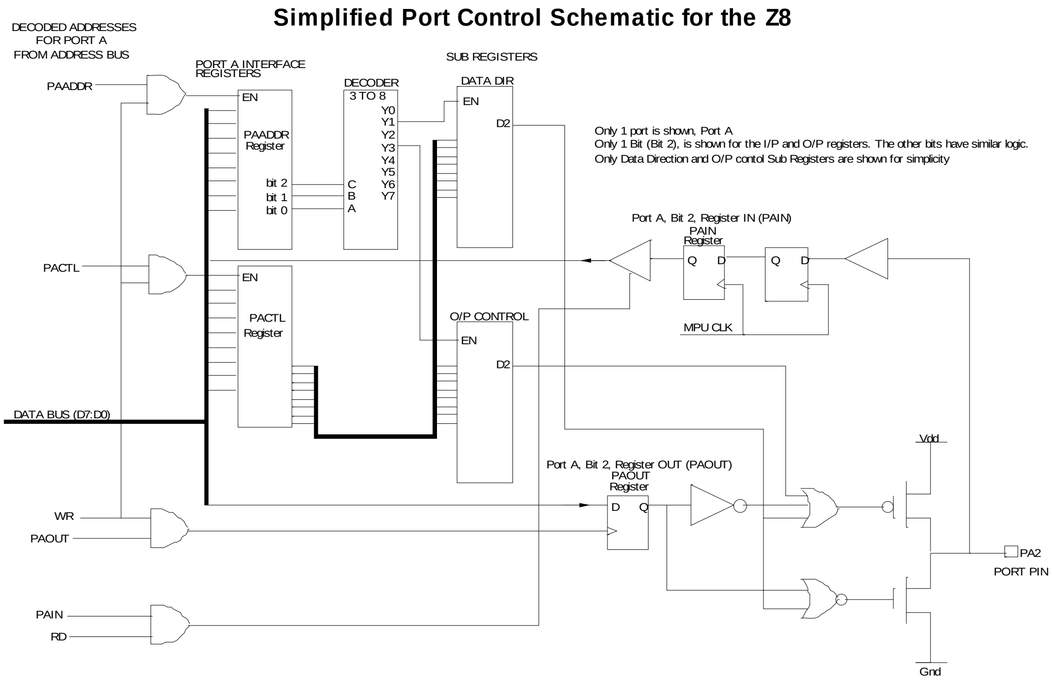
PAADDR and PACTL are used for configuring HOW Port A will work

4 Addresses are decoded for each Port

PAIN and PAOUT are used for communicating with the PORT A PINS

Z8 Address bus, Data bus and Control lines

Fig. 2



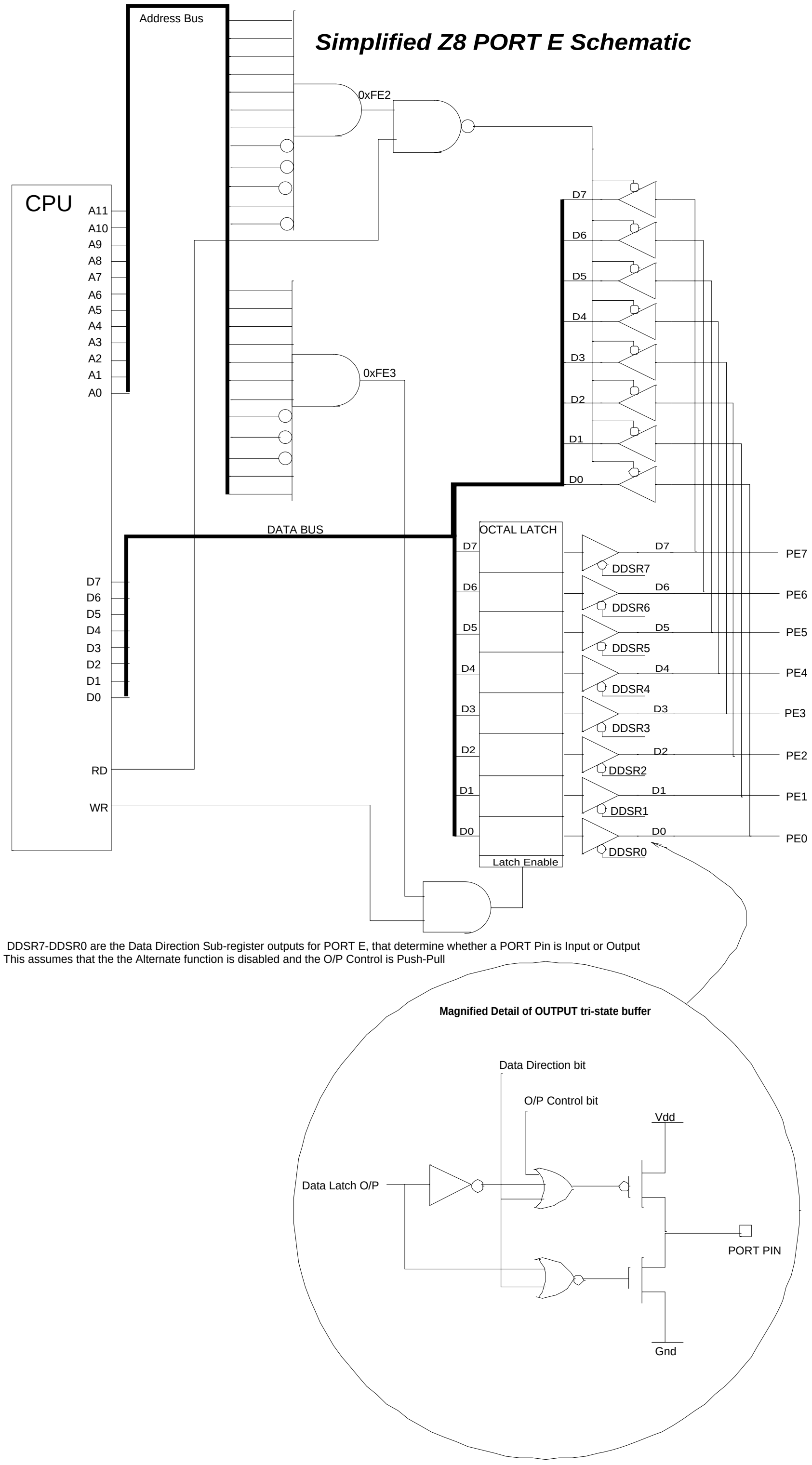
Simplified Port Control Schematic for the Z8

Only 1 port is shown, Port A
Only 1 Bit (Bit 2), is shown for the I/P and O/P registers. The other bits have similar logic.
Only Data Direction and O/P contol Sub Registers are shown for simplicity

Fig. 3: Z8 sub registers

Hex Value	Defined Constant	Function
0x01	DATA_DIR	// Data Direction sub register
0x02	ALT_FUN	// Alternate Function
0x03	OUT_CTL	// Output Control
0x04	HDR_EN	// High Drive enable
0x05	SMRS_EN	// Stop Mode Recovery Source Enable

Fig. 4: Z8 Simplified Z8 port E Schematic



DDSR7-DDSR0 are the Data Direction Sub-register outputs for PORT E, that determine whether a PORT Pin is Input or Output
This assumes that the the Alternate function is disabled and the O/P Control is Push-Pull

Unit 2

UNIT 2: PROG-2000 Introduction to Microcontrollers

Getting Started with the Z8 and C programming

Unit Objectives:

The student will be able to:

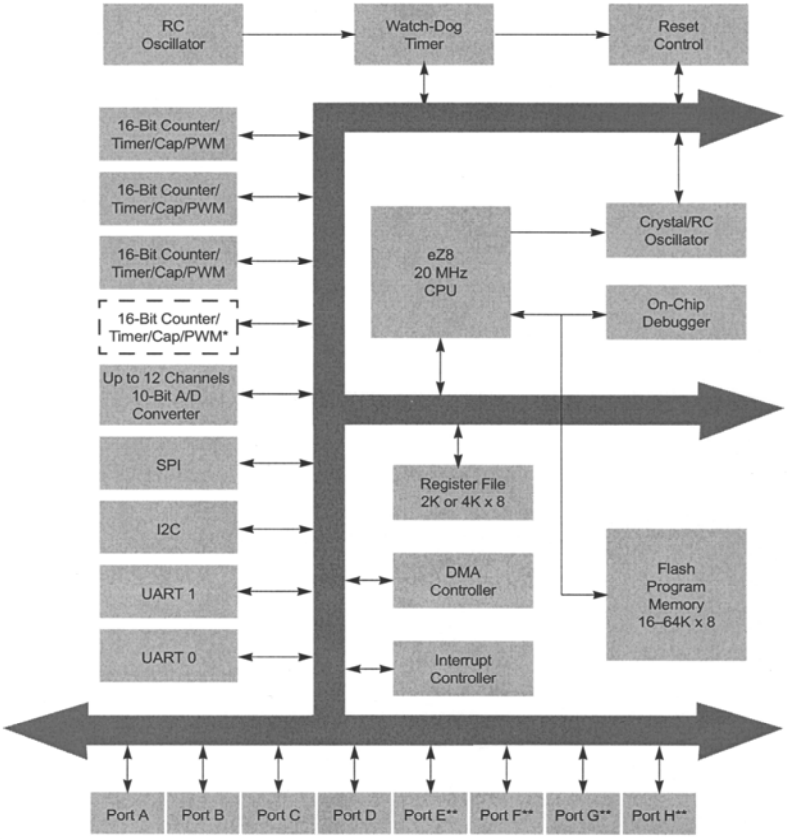
1. Configure the Z8 IDE *project settings*.
2. Use the Z8 IDE to download and run C programs on the Z8 MCU.
3. Describe the different parts of a C program from a program listing.
4. Describe ‘C’ data types, variables and constants and the size of each type.
5. Describe a *breakpoint* and its use.
6. *Single step* through a program.
7. Provide a descriptive comment for each line of a C program.
8. Examine the program counter (P.C.) address from the *register* window.
9. Configure the Z8 port registers using C.
10. Identify port pins on the Z8 schematic.
11. Describe the operation of the LED interface of the Z8 Microcontroller board.

In this unit student should do the following:

1. Try out your Z8 micro kit! Read and follow the steps in the Z8 Encore! Development Kit's “Quick Start Guide”. A printed version of this guide is in your kit and is also available as a PDF file on the included CD. The ZDS II software on the CD is ***already*** installed on the classroom computers, but you can install it on your home computer. You will also load and run a sample project from this CD. Follow the steps under the heading “Getting Started with ZDS II” starting on page 4 to do this.
2. Read chapters 2, 3 and 5. These are short chapters. ***Keep notes*** of the highlights! This will save you time when you need to review the concepts.
3. Read the handout “Making Projects with the Z8 Kit”. Follow these steps ***each time*** you make a new project. The “C source file” is the text that you type into the EDIT window. The source file for this lab will be called “lab2_1.c”. Remember to put it in a new subdirectory. This subdirectory name can be “lab2_1”. When you create the project use the ***same name*** as the directory. It will be given an extension of “.zdsproj”. ***Do not*** use spaces or punctuation in any file or directory names. Use the same format as described here for ***all*** your programs.
4. Do lab 2.
5. Examine the schematic of the Z8 Microcontroller board. Note that the 3 lsb of port A are connected to the LED cathodes of the Z8 LEDs. What logic level is required to turn on an LED? Can you change the value of the “leds” variable in program 2.1 to light different LEDs?
6. Answer the questions relevant for the unit 2 from the assignments package on a separate piece of paper. Some of these questions will be marked in class.
7. Show the instructor your Lab Manual and demonstrate your program’s operation.

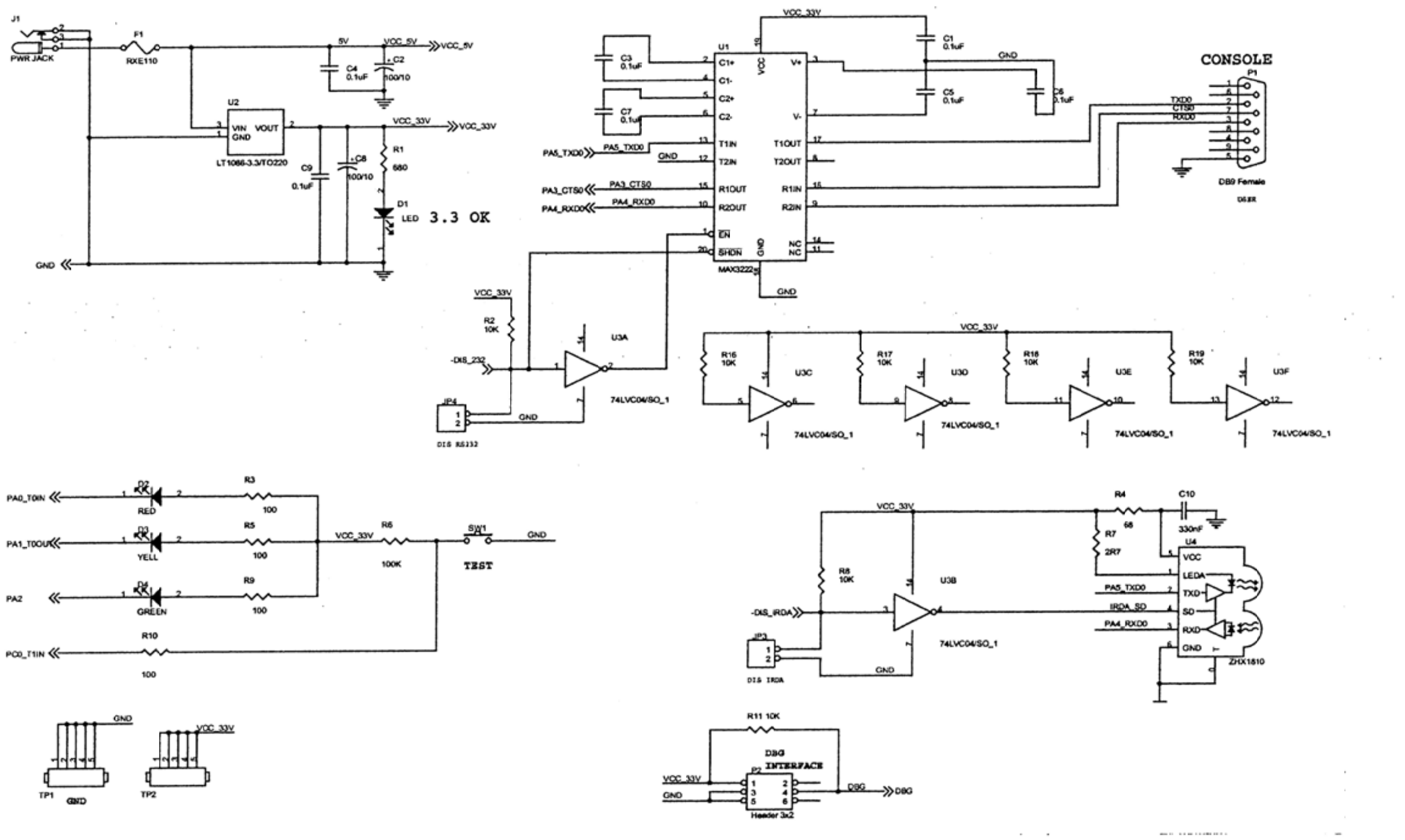
Architecture

Figure 1 illustrates the Z8F642x/Z8F482x/Z8F322x/Z8F242x/Z8F162x devices block diagram.

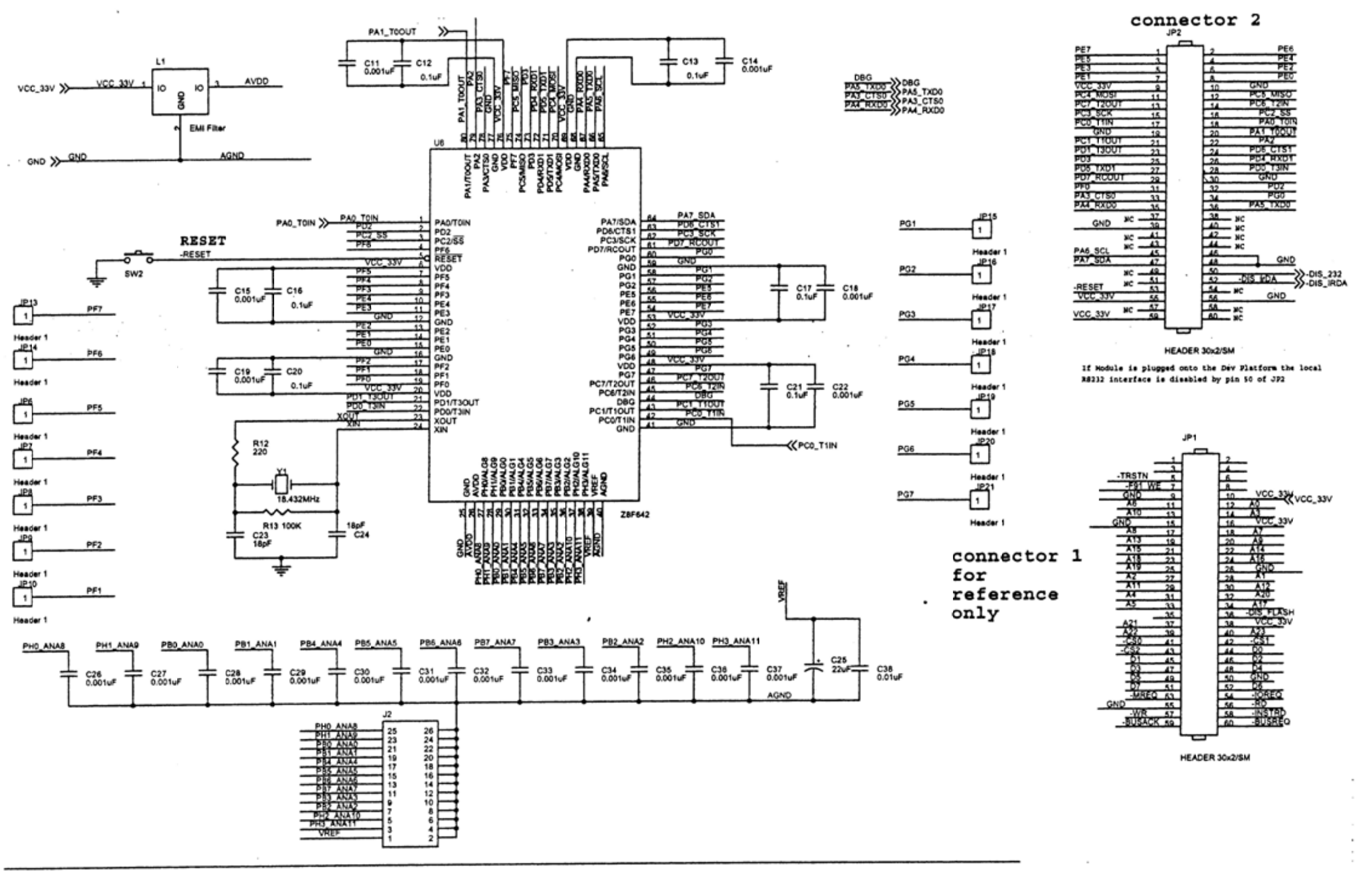

RC Oscillator
Watch-Dog Timer
Reset Control
16-Bit Counter/ Timer/Cap/PWM
16-Bit Counter/ Timer/Cap/PWM
16-Bit Counter/ Timer/Cap/PWM
16-Bit Counter/ Timer/Cap/PWM*
Up to 12 Channels 10-Bit A/D Converter
SPI
I2C
UART 1
UART 0
eZ8 20 MHz CPU
Crystal/RC Oscillator
On-Chip Debugger
Register File 2K or 4K x 8
DMA Controller
Interrupt Controller
Flash Program Memory 16–64K x 8
Port A
Port B
Port C
Port D
Port E**
Port F**
Port G**
Port H**

* The 40- and 44-pin packages feature only 3 timers. The fourth timer is unavailable in these packages.
** Ports E, F, G and H availability depends on the package.

Figure 1. Z8F642x/Z8F482x/Z8F322x/Z8F242x/Z8F162x Devices Block Diagram



Schematic, Z8 Encore!® 64K Series MCU Development Board, Page 1 of 2



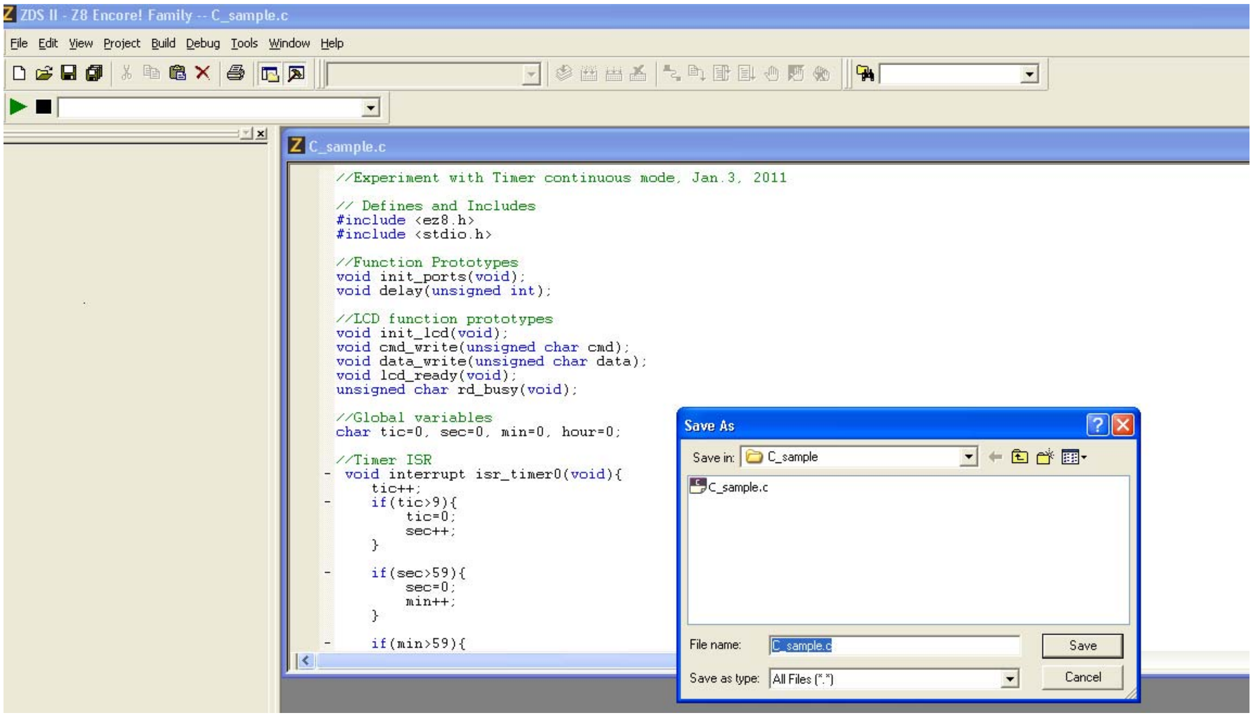
If Module is plugged onto the Dev Platform the local RS232 interface is disabled by pin 50 of JP2

Schematic, Z8 Encore!® 64K Series MCU Development Board Page 2 of 2

<u>Unit 2: PROG-2000, Making Projects with the Z8 Kit</u>

1. First, all projects must be in a separate directory, just like Quartus2. Create a directory and give it a meaningful name. Your primary file for your project, will be a "C source file " which is ascii text. This is the program that you write in the C language. To start writing a C program select ***File|Newfile***. Next use ***File|SaveAs*** to save the C source code file in your project directory. Use the same name as the project directory, except with the ".c" extension.
2. To create a project select ***File|NewProject*** and follow the Wizard's directions. Browse to your new directory and give the project the same name as your directory. After completing the Wizard, add the C file to your project using the ***Project|AddFiles*** option. Make sure that the correct CPU and CPU Family is selected. Verify that the specified directory is your project directory.
3. Now select the ***Projects|Settings*** menu option.
    - In the "Code Generation" option, make sure that parameters are passed by memory.
    - In the "Debugger" option, select the "USB smart Cable" as the Debug Tool.
    - Examine the screen shots below, just to make sure that you choose the correct options.
4. Always save your project.
5. Follow these steps whenever you are writing your C programs. Refer to page 7 of the "Quickstart guide" for building, downloading and running your code on the Z8.

<u>Screenshot Examples</u>


ZDS II - Z8 Encore! Family -- C_sample.c
File Edit View Project Build Debug Tools Window Help
C_sample.c
Save As
Save
Cancel

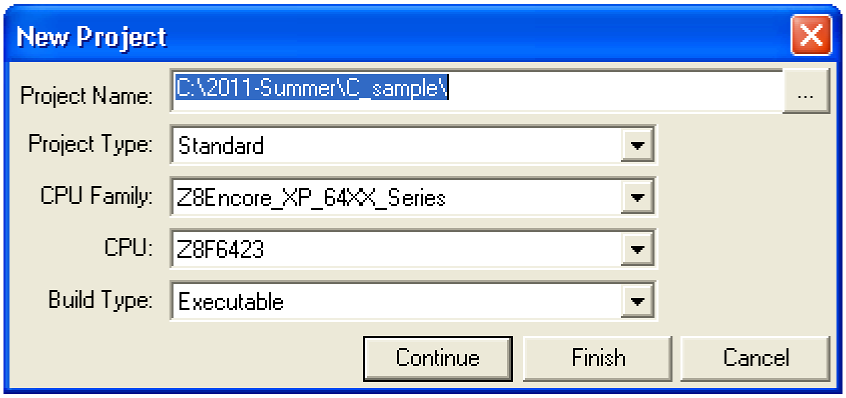
```
//Experiment with Timer continuous mode, Jan.3, 2011

// Defines and Includes
#include <ez8.h>
#include <stdio.h>

//Function Prototypes
void init_ports(void);
void delay(unsigned int);

//LCD function prototypes
void init_lcd(void);
void cmd_write(unsigned char cmd);
void data_write(unsigned char data);
void lcd_ready(void);
unsigned char rd_busy(void);

//Global variables
char tic=0, sec=0, min=0, hour=0;

//Timer ISR
 void interrupt isr_timer0(void){
    tic++;
    if(tic>9){
        tic=0;
        sec++;
    }

    if(sec>59){
        sec=0;
        min++;
    }

    if(min>59){
```

Save in: C_sample

C_sample.c

File name: C_sample.c

Save as type: All Files (*.*)



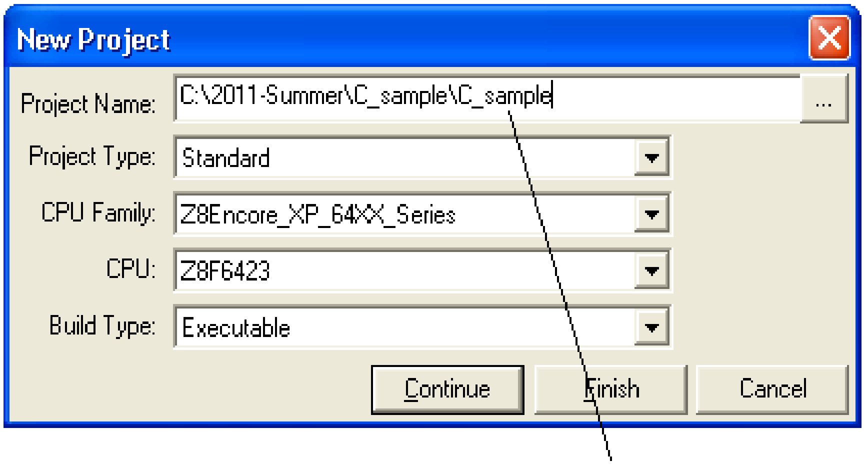
Project Name: C:\2011-Summer\C_sample\

Project Type: Standard

CPU Family: Z8Encore_XP_64XX_Series

CPU: Z8F6423

Build Type: Executable



Project Name: C:\2011-Summer\C_sample\C_sample

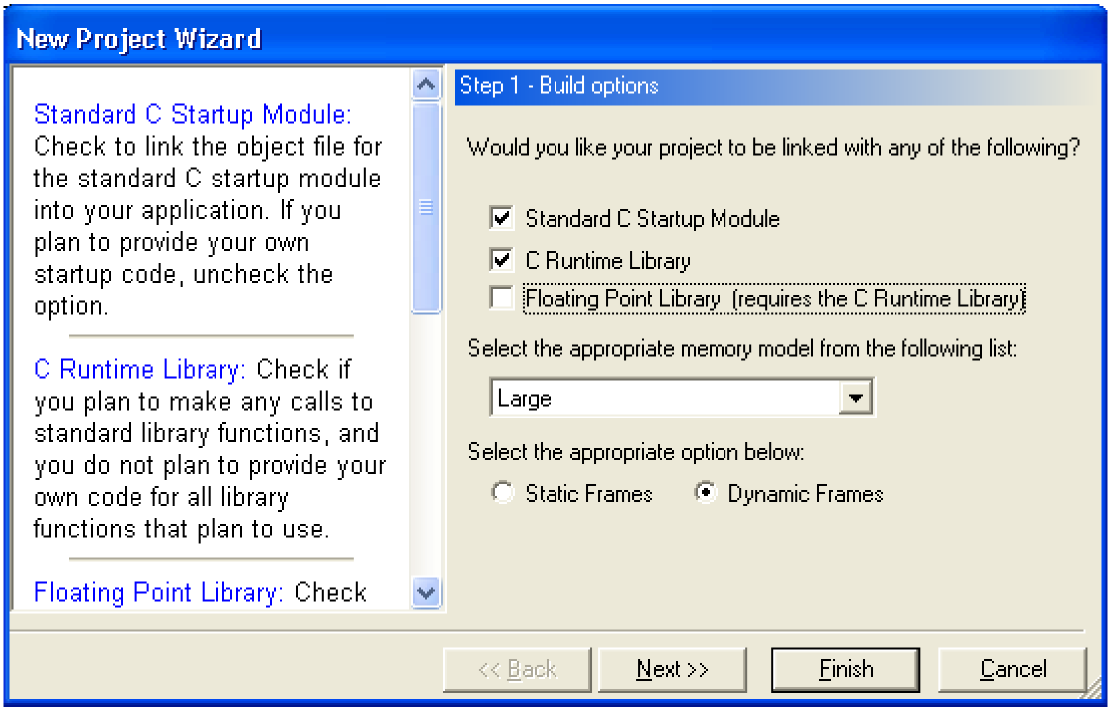
Project Type: Standard

CPU Family: Z8Encore_XP_64XX_Series

CPU: Z8F6423

Build Type: Executable

Type a Project name in the Project directory. This project name wil automatically be given an extension of ".zdsproj". Click on the "Finish" button when you are done.



New Project Wizard

Standard C Startup Module: Check to link the object file for the standard C startup module into your application. If you plan to provide your own startup code, uncheck the option.

C Runtime Library: Check if you plan to make any calls to standard library functions, and you do not plan to provide your own code for all library functions that plan to use.

Floating Point Library: Check

Step 1 - Build options

Would you like your project to be linked with any of the following?

☑ Standard C Startup Module

☑ C Runtime Library

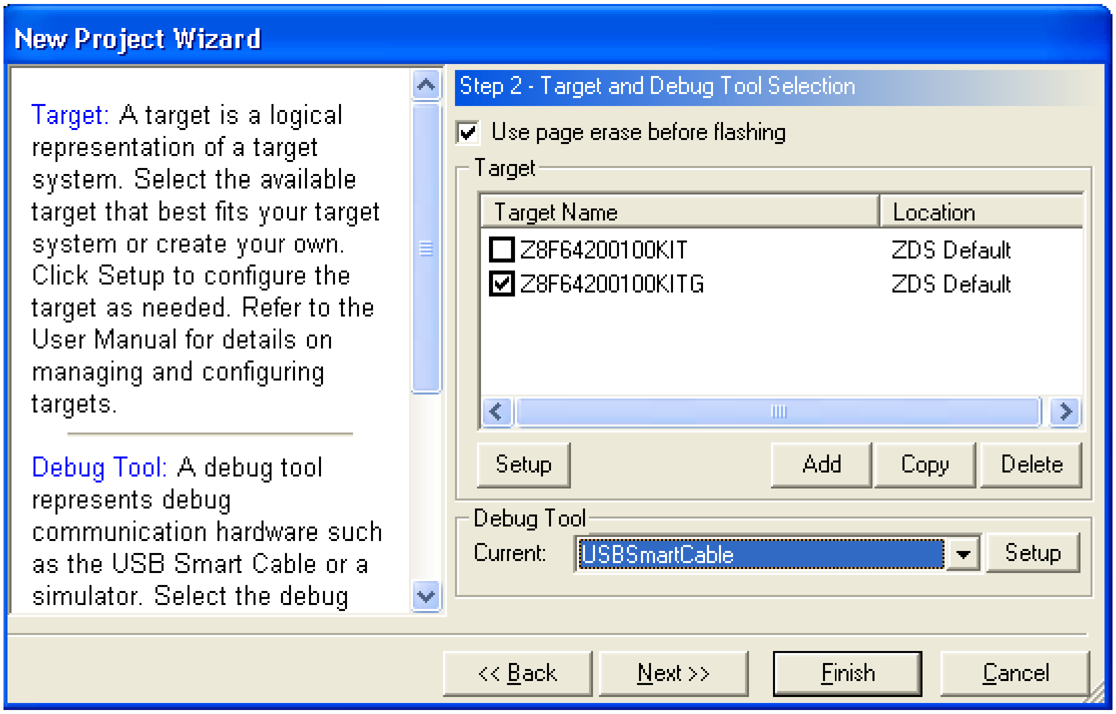
□ Floating Point Library (requires the C Runtime Library)

Select the appropriate memory model from the following list:

Large

Select the appropriate option below:

○ Static Frames ● Dynamic Frames



New Project Wizard

Target: A target is a logical representation of a target system. Select the available target that best fits your target system or create your own. Click Setup to configure the target as needed. Refer to the User Manual for details on managing and configuring targets.

Debug Tool: A debug tool represents debug communication hardware such as the USB Smart Cable or a simulator. Select the debug

Step 2 - Target and Debug Tool Selection

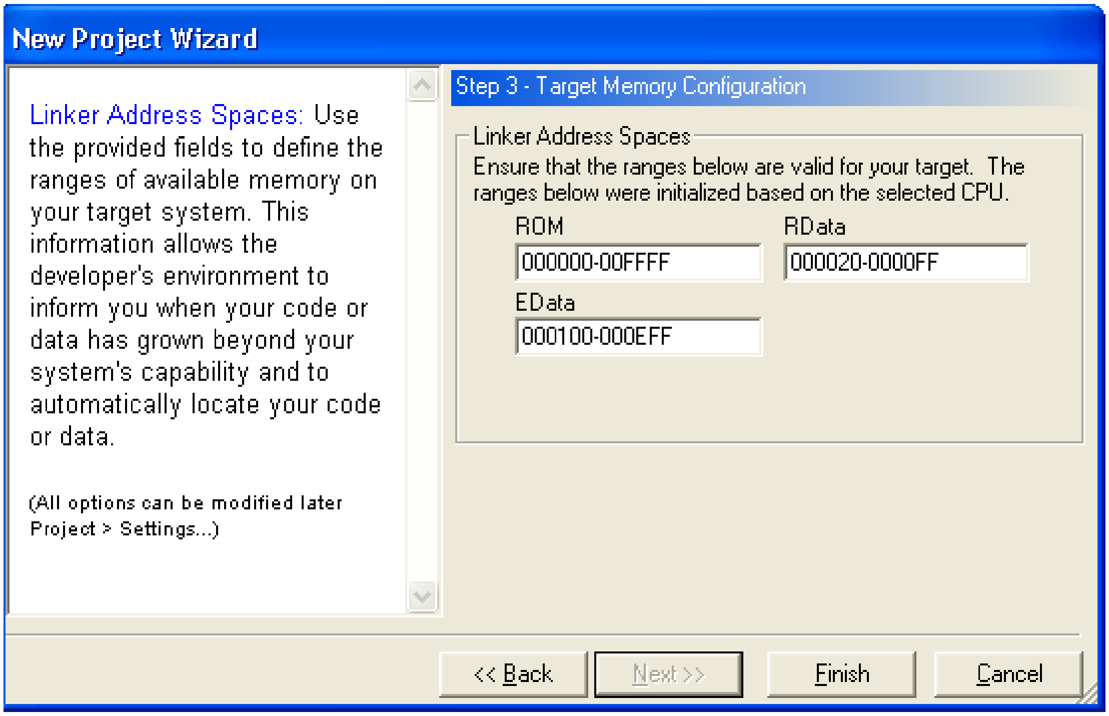
☑ Use page erase before flashing

Target

Target Name	Location
□ Z8F64200100KIT	ZDS Default
☑ Z8F64200100KITG	ZDS Default

Debug Tool

Current: USBSmartCable



New Project Wizard

Linker Address Spaces: Use the provided fields to define the ranges of available memory on your target system. This information allows the developer's environment to inform you when your code or data has grown beyond your system's capability and to automatically locate your code or data.

(All options can be modified later Project > Settings...)

Step 3 - Target Memory Configuration

Linker Address Spaces

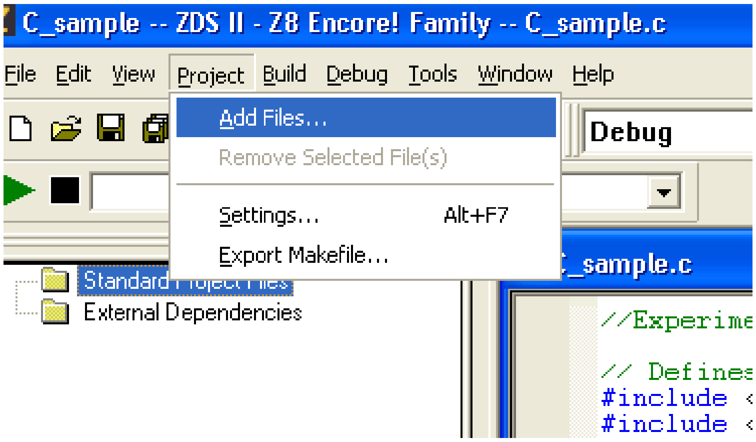
Ensure that the ranges below are valid for your target. The ranges below were initialized based on the selected CPU.

ROM: 000000-00FFFF

RData: 000020-0000FF

EData: 000100-000EFF

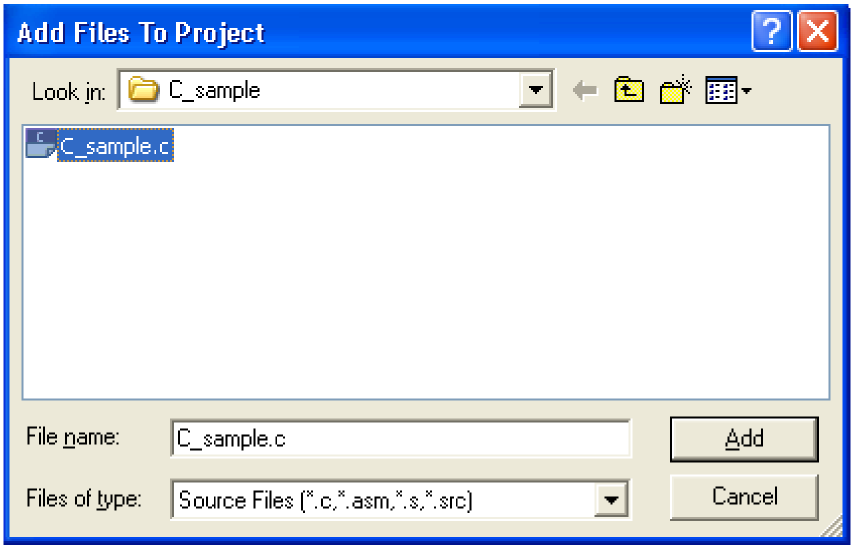
Click on Finish.



C_sample -- ZDS II - Z8 Encore! Family -- C_sample.c

```
//Experime
// Defines
#include <
#include <
```

Select the Project Menu as shown, then select "Add Files..."



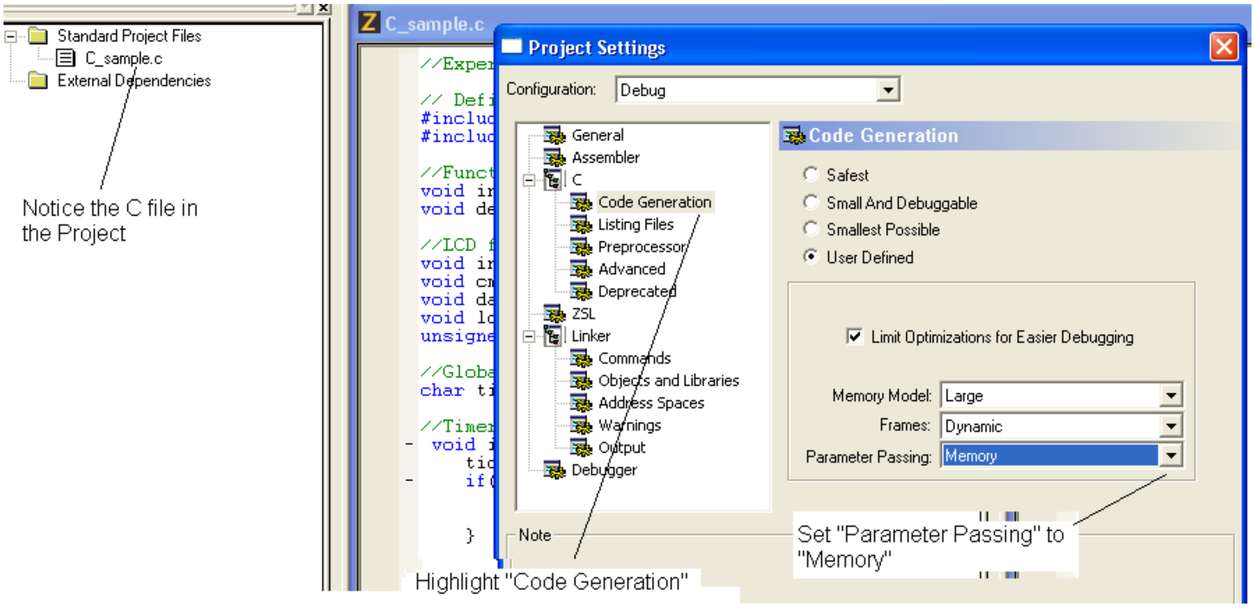
Look in: C_sample

File name: C_sample.c

Files of type: Source Files (*.c,*.asm,*.s,*.src)

Highlight the C file, the Click the "Add" button.

Now, Select the Projects Settings Menu option:



Notice the C file in the Project

Configuration: Debug

○ Safest
○ Small And Debuggable
○ Smallest Possible
● User Defined

☑ Limit Optimizations for Easier Debugging

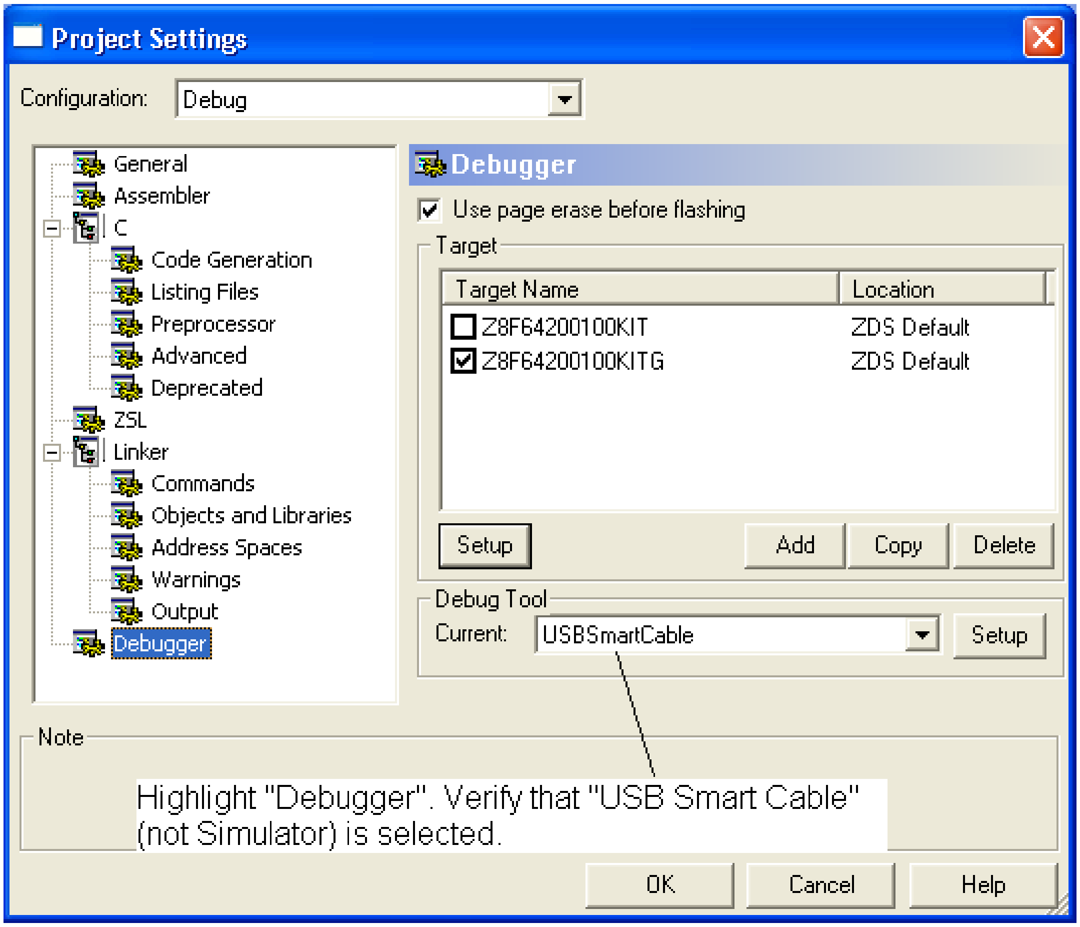
Memory Model: Large

Frames: Dynamic

Parameter Passing: Memory

Highlight "Code Generation"

Set "Parameter Passing" to "Memory"



☑ Use page erase before flashing

Target Name	Location
□ Z8F64200100KIT	ZDS Default
☑ Z8F64200100KITG	ZDS Default

Current: USBSmartCable

Highlight "Debugger". Verify that "USB Smart Cable" (not Simulator) is selected.

Unit 2: My stupid Z8 doesn't work! What do I do now?

First of all, let me assure you that your Z8 is not stupid. Since you cannot attribute intelligence to a programmed controller, it is not capable of being stupid. Humans on the other hand, have intelligence. Logically, therefore...

Things to check:

1. With regard to the "Quick Start Guide", the ZDS software is already installed on the computers in the classroom, so you can just run the software. The ZDSII software is located in the "Programming Apps" folder in the START MENU.
   - The path for the demo project that is: C:\Programfiles\ZiLOG\ZDSII_Z8Encore!_4.10.1\samples\Z8F642x_ledBlink\src\ledblink.zdsproj
   - Look at the big chip on your development board (the Z8). Under "Zilog" you should see a part number "Z8F". On the next line you should see "6423FT020SC". This indicates that your chip is a Z8F623. The project directory above works with your chip (Z8F642x). The x just means any version of this chip that starts with those first 5 characters will run this project correctly.

2. Make sure that you follow the steps as outlined in this Unit. That means creating a directory for your project (same as a folder) that holds the files for your project. Make the directory name the same as your file name except that the directory name has no ".c extension. The CPU family is the "64K series". ***DO NOT use spaces or punctuation in your file names***.

3. Remember that All projects files have the extension of ".zdsproj". This extension is added automatically. All C text files have the extension ".c". ***You must add*** the C source file to your project, after creating your project with the Project Wizard.

4. Most of you will have a "USB Smart Cable". Select the correct interface for your board. You can do this in the "Debugger" page of the "Project Settings" window (select the Projects menu, then Settings). In the class room the USB driver will have to be installed each time you plug in your USB adapter. When it offers to select the driver, browse to the following path: "C:\drivers\Zilog XTOOLS" and install the driver.

5. For Lab 2 verify the following using Windows Explorer:
   - A directory called "lab2_1". This is your project directory, where all project files are located.
   - lab2_1.c is the 'C' source file (text) created in the ZDSII editor. Make sure that it has a ".c" extension.
   - lab2_1.zdsproj is the name of the project file.

UNIT 2: PROG-2000 Introduction to Microcontrollers

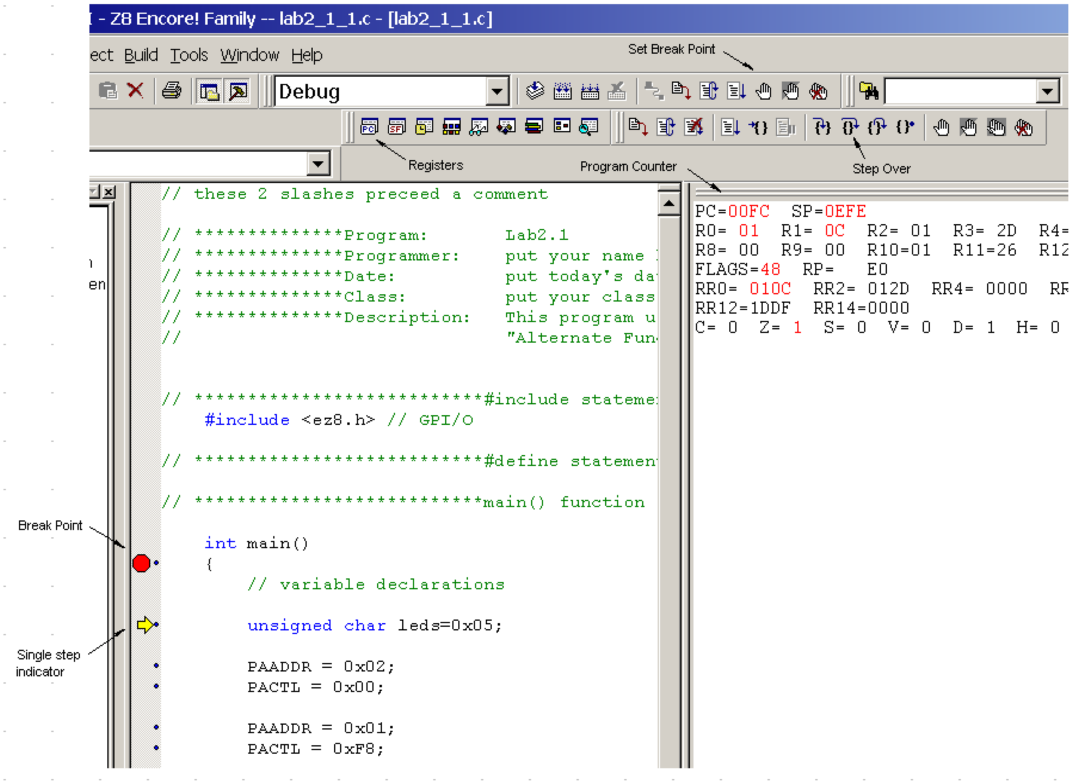
Lab 2 (call this program “lab2_1.c”)

1. Type in the following C source file into an edit window in ZDS II:

```
// ***************Program:               Lab 2.1
// ***************Programmer:            put your name here!
// ***************Date:                  put today's date here!
// ***************Class:                 put your class here!
// ***************Description:           This program uses the 3 lsb's of Port A for O/P only
//                                       "Alternate Function" is disabled
// these 2 slashes precede a comment

// *****************************#include statements
        #include <ez8.h> // GPI/O

// *****************************#define statements

// *****************************main() function

  int main(void)
  {
              // variable declarations

              unsigned char leds=0x05;

              PAADDR = 0x02;
              PACTL = 0x00;

              PAADDR = 0x01;
              PACTL = 0xF8;

              PAADDR = 0x03;
              PACTL = 0x00;

              PAADDR = 0x00; // prevents inadvertent changes to sub registers

                      PAOUT = leds;

              return 0; // Finished!
              }
//*******************************end of program
```

2. After creating a project in a new directory, build the program and execute it (use the Icons: ***Save Project***, ***Build All***, ***Reset***, ***Go***, in that order). Wow! Your program runs. If it doesn’t, check for errors listed in the message window at the bottom of the screen. Often syntax errors, such as a misspelled word or a missing semi-colon, are the problem.
3. Note what happens to the LED's on the Z8 board, after running the program.
4. To start over do the following: select **Build | Debug | Stop Debugging** from the main menu.

5. Next, put your cursor on the same line as the opening brace in the main() function. From the top toolbar select the hand icon (set break point). A red dot should appear at that spot. A breakpoint stops the program execution at that point. Look at the diagram below to locate the various icons.
6. Select the ***Reset*** icon. The yellow arrow should point to the statement just below the breakpoint. Click on the “Step Over” icon (or F10). Did you notice how the yellow arrow moves down to the next C statement? Each time you “Step over” you are executing the next C statement. This is called “single stepping” through your program.
7. Click on the first icon in the second row (you will see a “PC” in the icon). This icon is called “Registers”. Selecting it will open up a window that displays the CPU's internal registers including the ***Instruction Pointer*** (here called the Program Counter or PC). Watch how the value of PC changes as you single step through the program. It will always display the address of the next instruction to be executed.
8. At each “step” of the program, ***note*** what is happening on the LEDs. Stop single stepping when the yellow arrow is on the closing brace of the program.
9. Add a clear, understandable ***comment*** next to each C statement, in your source code, then place a copy of lab2_1.c in your Lab Manual. Identify each statement that causes a change in the state of the LEDs. ***Explain*** the events that you have observed, to your instructor.



```
// these 2 slashes preceed a comment

// **************Program:         Lab2.1
// **************Programmer:      put your name
// **************Date:            put today's da
// **************Class:           put your class
// **************Description:     This program u
//                                "Alternate Fun


// ***************************#include stateme
    #include <ez8.h> // GPI/O

// ***************************#define statemen

// ***************************main() function

    int main()
    {
        // variable declarations

        unsigned char leds=0x05;

        PAADDR = 0x02;
        PACTL = 0x00;

        PAADDR = 0x01;
        PACTL = 0xF8;
```

```
PC=00FC   SP=0EFE
R0= 01   R1= 0C   R2= 01   R3= 2D   R4=
R8= 00   R9= 00   R10=01   R11=26   R12
FLAGS=48   RP=    E0
RR0= 010C   RR2= 012D   RR4= 0000   RF
RR12=1DDF   RR14=0000
C= 0   Z= 1   S= 0   V= 0   D= 1   H= 0
```

Unit 3

UNIT 3: PROG-2000 Introduction to Microcontrollers

Program Flow Control and Bitwise operators

Unit Objectives:

The student will be able to:

1. Distinguish between the correct and incorrect use of C syntax and symbols.
2. Describe arithmetic, **relational** and **logical** operators in C programs.
3. Read data from an I/P port using C.
4. Write data to an O/P port using C.
5. Write programs using **bitwise** "and", "or", "xor" and "complement" operators.
6. Describe the term "bit masking" and its purpose.
7. Distinguish between the assignment operator and the comparison operator.
8. Identify and describe the "flow control" statements in a program.
9. Describe the "test" switch interface on the Z8 Microcontroller board using the schematic.

In this unit student should do the following:

1. Read "C by Example", chapters 7, 8, 9 and p.157 to 162 of chapter 10.
2. Read the handout: "Bit operations in C".
3. Do lab 3.1, including the lab questions. Put the commented source code for this lab in your Lab Manual (a duo-tang folder) and show your instructor.
4. Demonstrate and explain the operation of lab3_1.c to your instructor.
5. Learn more about loops. Read the rest of chapter 10 and chapter 11. Don't forget to make notes on what you have read.
6. Answer the questions relevant for the unit 3 from the assignments package on a separate piece of paper. Some of these questions will be marked in class.

UNIT 3: PROG-2000 Introduction to Microcontrollers

Notes: Bit Operations in C

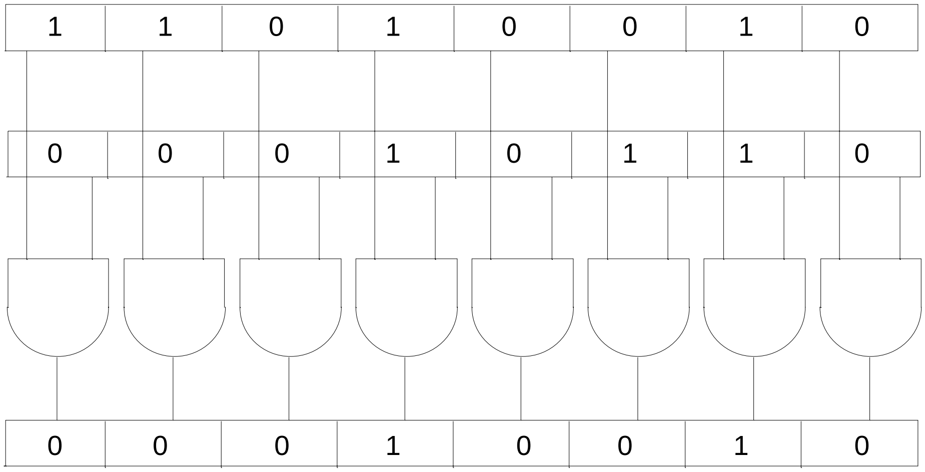
Bitwise operator: This is an operator that performs a logical operation on individual bit positions in a binary number. Bitwise operators work on integers and characters that are either constants or variables. They can be signed or unsigned.

These are the operators we will consider:

- &   - Bitwise "and"
- | - Bitwise "or"
- ^   - Bitwise "exclusive or"
- ~   - complement bits
- <<   - shift left
- >>   - shift right

It is assumed that you know the rules for logical operations.

In a byte, the most significant bit position is bit 7 (left - most bit) and the least significant is bit 0. When performing an "and" on two different bytes, each corresponding bit position is effectively applied to one input of a 2 input "and" gate. This is illustrated in the diagram below.



Each byte is referred to as an operand. One operand can be used to "mask" or cover up the corresponding bits of the other operand (the data), ***if*** the appropriate bits are '0'. In the example above: 0xD2(the data) **&** 0x16(the mask) = 0x12. Any bit in the ***mask*** that is a 0 disables the gate and the O/P must be 0 no matter what logic level is on the other I/P. Looking at it another way, any bit that is masked is cleared to zero. Therefore this operator is used to clear specific bits to a logic '0'.

The inclusive "or" (or bitwise "or") is used to force specific bits to '1' in a way that is similar to how an '&' operation clears bits, on a bit by bit basis. Use it to set bits to 1. For example: **val=val|0x03;** //forces 2 lsb’s of val to logic 1.

The complement operator (~) inverts individual bits of the operand.

The "exclusive or" (^) causes no change to bits that are "XOR ed" with , '0' but complements bits that are "XOR ed" with '1'. It's like a programmable inverter. Remember that all these operators work on corresponding bit positions.

Shift left (<<) and shift right (>>) operators shift the bits that make up the operand, the appropriate number of bit positions.

ie. code = code << 2

If the variable "code" originally contained 01101110 in binary , after shifting two places left , it would contain 10111000. This is B8 in Hex , but it could be -72 decimal or 184 decimal , depending on whether the variable "code" was declared as signed or unsigned. If a variable is signed, the most significant bit of the byte or 16 bit word is a sign bit. If this bit is a '1' the result is negative and in 2's compliment form (see page 470, Appendix A for a discussion of negative binary numbers). If the MSB is '0' the number is positive and is determined by taking the sum of the binary place values.

ie. 01010110 = 64 + 16 + 4 + 2 = 86 decimal.

If a variable is unsigned , the value is represented by the sum of the binary weights of all bits. Declaring a variable as signed or unsigned has implications for bitwise shifting. Let's shift a short integer variable 3 places to the right.

value = value >> 3

If ***value*** was signed : 11010110 is equal to -42 decimal. If ***value*** was unsigned: 11010110 is equal to 214 decimal.

After shifting 3 places right: signed = 11111010 = -5 decimal, unsigned = 00011010 = 26 decimal. Notice that the value declared as signed will shift a copy of the sign bit (either a 1 or a 0) to subsequent positions but an unsigned value ***always*** shifts 0's from the left.

For a signed number the right shift is an "arithmetic shift" (the sign bit matters), but for an unsigned value it is known as a "logical shift". Left shifts always fill in the vacated positions with 0's.

Here is an example of how shifts and logical operators can be used. This code fragment will check to see if bit 3 is logic '1'.

```
int value = 0x45; // 45 is declared as a hex constant
value = value >> 3; // shift right 3 places
value & = 0x0l; // another way of saying: value = value & 0x01
if (value == 0)
        printf(" Bit 3 is cleared");
else printf(" Bit 3 is set");
```

That's it. We won't be using the shift operations until we write code to scan a matrix keypad, but try your best to learn how to use ***all*** the bitwise operators. Write some simple programs and look at your variables in the watch window.

UNIT 3: PROG-2000 Introduction to Microcontrollers

Lab 3

1. Type in the C source file on the next page, into an edit window in ZDS II.
2. Fill in the information header at the top of the program.
3. Analyze the program. Type in a ***descriptive comment*** next to each C statement, explaining the ***purpose*** of the statement.
4. Create a project, build and execute it.
5. Provide the answers for the questions below and put them, along with your source code, in your Lab Manual as Lab 3. Show your instructor your Lab Manual.
6. Demonstrate the program to your instructor, while explaining its operation.

Lab 3 Questions:

1. Examine the Z8 Microcontroller board schematic. Identify the test switch interface. Which port pin is connected to the test switch? What logic level is generated by the test switch when it is pressed?
2. Running the program at full speed, push the "test" push button several times. What did you observe?
3. Which bit of which port is used to sense the logic state of the "test" switch?
4. Stop debugging and "single step" through the program. After you notice a repetitive pattern, hold in the "test" Push Button while you single step. Continue to single step with the test button pushed, then, repeat without pushing the test push button. Can you explain this behavior?
5. What bitwise logical operations are performed in this program?
6. Explain the function of bit masking, as used in this program.
7. Does this program stop by itself? Why?
8. How does "=" differ from "=="?
9. What is a "compound statement"?

```
// **************Program:                    lab3_1.c
// **************Programmer:                 put your name here!
// **************Date:                       put today's date here!
// **************Class:                      put your class here!
// **************Description:                fill this in!


// ***************************#include statements
#include <ez8.h> // GPI/O definitions

// ***************************#define statements

// ***************************main() function
int main(void)
   {
//variable declarations and assignments;
         unsigned char leds=0x02;
         unsigned char push_button;
         push_button=0x00;

//initialize port A
         PAADDR=0x02;
         PACTL=0x00;

         PAADDR=0x01;
         PACTL=0xF8;

         PAADDR=0x03;
         PACTL=0x00;

         PAADDR=0x00; // prevents inadvertent changes to sub registers

//initialize port C
         PCADDR=0x02;
         PCCTL=0x00;

         PCADDR=0x01;
         PCCTL=0x01;

         PCADDR=0x00;

//light LEDs
         PAOUT=leds;
         while (1){
                  push_button=PCIN;
                  push_button=push_button & 0x01;
                           if(push_button==0x00){
                                    leds=leds^0x07;
                                    PAOUT=leds;
                           }
         }
return 0;
}
//*****************************end of program
```

Unit 4

UNIT 4: PROG-2000 Introduction to Microcontrollers

Arrays and Pointers

Unit Objectives:

The student will be able to:

1. Distinguish between integer and character arrays.
2. Describe how an array of characters differs from a string array.
3. Describe how both an array and a pointer can be used together.
4. Distinguish between an address of a variable and a variable.
5. Use the “address of” and “dereferencing” operators in a program.
6. Use both array subscripts and pointers, to access array elements.
7. Use pointer arithmetic to modify pointer values.
8. Copy the contents of one array to another.
9. Use the “watch” window to examine variables while debugging.
10. Describe a program’s operation using pseudo-code.

In this unit student should do the following:

1. Read “C by Example”, chapters 4, 19 (p.324-334), 20 and 21. These chapters discuss arrays and pointers.
2. Read the handout on “Pseudo-code”.
3. Answer the questions relevant for the unit 4 from the assignments package on a separate piece of paper. Some of these questions will be marked in class.
4. Do Lab 4, including questions 8.1 to 8.5. Place your pseudo-code, source code and answers to the questions in your Lab Manual. Show your instructor.
5. Demonstrate your functioning program to your instructor and explain its operation.

Using Pseudocode

Developers currently can choose from a variety of design methodologies. Many of them, at some point, have you write all or part of your program in **pseudocode**. When we write statements in pseudocode, we write generic, high-level statements of program actions.

Pseudocode provides a language-independent of describing what a program should do. In this case the term "language-independent" refers to computer languages such as Pascal or C, not human languages like English or Russian.

Our goal is to start with a clear problem statement, develop an **algorithm** that solves the problem, and end with a program that implements the algorithm. An algorithm is step-by-step solution to a problem. It is similar to a recipe for cooking food. A recipe for strawberry cream pie is the algorithm that we can follow to turn raw ingredients into a great dessert.

The first step in developing a program is to identify the problem or task, We'll use the example of a program that finds the average of three numbers. The statement of purpose for the program is shown in Example 5.2.

EXAMPLE 5.2 Statement of Purpose

```
Find and print the average of three numbers.
```

The second step is to understand the problem. To find the average of three numbers, we must add the numbers and divide the result by three. Example 5.3 is an initial pseudocode version of the program.

EXAMPLE 5.3 The Initial Pseudocode

```
Find the average of three numbers.
    Add the three numbers.
    Divide the sum by 3.
Print the resulting average.
```

The third and fourth steps for solving a problem are to identify alternative ways to solve the problem and to select the best way to solve the problem from the list of alternative solutions. Because this problem is so simple, we don't really need to perform these two steps. However, the ability to perform these two steps is crucial to being able to provide software that meets our users' needs.

The next steps in developing this program are to list instructions that will enable us to solve the problem and to evaluate the solution. This is exactly what pseudocode does for us. The initial versions of the pseudocode provide us with a detailed problem statement. As we repeatedly evaluate and refine our pseudocode, we get the actual instructions we can use to write the program from. Example 5.4 shows a refinement of our pseudocode.

EXAMPLE 5.4 Refining the Algoritm

```
Get three numbers from the user.
Find the average of the three numbers.
    Add the three numbers.
    Divide the sum by 3.
Print the resulting average.
```

This pseudocode algorithm is good, but it is still not specific enough to write a program from. The algorithm now states where the numbers come from, but to not say how they are obtained from the user. Example 5.5 fixes this.

EXAMPLE 5.5 Specifying the Input Process

```
Get three numbers from the user.
    Prompt the user for an integer.
    Read an integer from the keyboard.
    Prompt the user for an integer.
    Read an integer from the keyboard.
    Prompt the user for an integer.
    Read an integer from the keyboard.
Find the average of the three numbers.
    Add the three numbers.
    Divide the sum by 3.
Print the resulting average.
```

With this refined algorithm, we can now develop the final pseudocode for our program, as shown in Example 5.6.

EXAMPLE 5.6 The Final Pseudocode Algorithm

```
Declare the integer variables num1, num2, num3, sum, and average.
Get three numbers from the user.
    Prompt the user for an integer.
    Read an integer from the keyboard. Store it in num1.
    Prompt the user for an integer.
    Read an integer from the keyboard. Store it in num2.
    Prompt the user for an integer.
    Read an integer from the keyboard. Store it in num3.
Find the average of the three numbers.
    sum = num1 + num2 + num3.
    average = sum / 3.
Print average to the screen.
```

Using this algorithm, we can write a program. Hands On 5.4 implements the algorithm in Example 5.6.

Common Errors with C Pointers

```
int * p_array, array[5]={0,1,2,3,4};//declarations
p_array=array[5]; //no go
*p_array=array[5]; //no go-outside array bounds
p_array=array[0]; //no go, put coffee in your coffee cup (putting data into an address variable or pointer)
*p_array=array; //no go, put gas in your gas tank (putting an address into a data variable)
*p_array=&array[0]; //no go, (putting an address into a data variable)
*p_array=array[0]; //OK, int data to an int variable
p_array=array; //OK, address to a pointer variable
p_array=&array[0]; //OK, address to a pointer variable
```

```
// Don’t put coffee in your gas tank (data in a pointer variable) or visa versa
```

UNIT 4: PROG-2000 Introduction to Microcontrollers

Lab 4

1. Type in the following code, putting everything in its proper place in the ***style template***. Don't forget to fill in the information header. Make a project and clean up any syntax errors. ***Read your text*** for further information on character arrays and pointers: chapter 4, chapter 20 and chapter 21.

```
// lab4_1.c
#include <ez8.h>
#include <stdio.h>
int main(void)
{
        char val;
        int cnt=0;
        char mystr[20]= “hello there!”;
        char *p_str2, str2[20] = “zzzzzzzzzzzzzzzz”;
        char* p_mystr=mystr;
        p_str2=str2;
                while (*p_mystr!=0x00){
                        val=*p_mystr++;
                        cnt++;
                }
return 0;
}
```

2. If you run this program at full speed it won't show you anything, but don't assume it doesn't ***do*** anything. In order to see what happens you will open a Watch Window and examine the variables as you single step through the statements. First, find “Watch Window” from the ***Help Index*** and read up on how to display the contents of variables in different ways: as ***decimal*** values, as ***hex*** values, as ascii ***characters*** or as ascii ***strings*** (asciz).

Put the following variables in the Watch Window (Only the variables. ***Not*** the comments):

- mystr   // mystr is a pointer ***constant***. This address cannot change
- asciz mystr   // the string will be displayed
- ascii mystr[0]   // the data in the first element of the character array
- ascii mystr[cnt]   // the data in the array element defined by cnt
- ascii (*p_mystr)   // p_mystr is a pointer ***variable.*** *p_mystr means the data in the memory location addressed by p_mystr
- p_mystr   //the address contained in p_mystr. Watch it change as you step.

3. Now it's your turn to “play computer”! On a separate piece of paper explain what you ***expect*** this program to do, step by step. What value should each variable display after each statement? Write an explanation of each statement and the variable values after each statement is executed. Do at least 2 iterations through the loop or until you understand what is happening.

4. After you ***think*** you understand what the program is doing, its time to single step through the program and examine what really ***does*** happen. Were you close?

5. Look up the Hex values of the array elements of “mystr” (click on the '+' sign to see them). How do they compare with the ascii code values in Appendix C?

6. **Programming Assignment (lab4_2.c):** Add new capabilities to program 4.1. Copy the string constant from the character array called “mystr” to a new array called “str2”, using ***pointers***.

   - To help you ***understand*** the program’s operation, draw a picture of the array and the pointer (look at fig.4.3 on p.57 of your text). Move the pointer through the array and visualize what you want to do.
   - Once you understand how to perform the copy operation, use ***pseudo-code*** to describe the actions of the program. This will prepare you, for writing the actual program in C. Remember that strings must be copied character by character (see the ***caution***, on p. 58 of the text).
   - Finally write your C source code, and debug it. Remember to put “str2” and any other variables you use, in the ***watch*** window. This way, you can see what you are doing and verify that the characters copy correctly. Comment your source code.

7. Print out the properly commented code including your information header and put it into your Lab Manual (the duo-tang folder that holds all your labs). Properly label each item. For example, your program assignment would be labeled “lab4_2.c”. **Include the answers to Questions 8.1 to 8.5** in your lab manual as well. Don't forget your name, class and date on the top of each lab assignment! ***Show your instructor your work!***

8. Questions to answer:

   8.1 How does a pointer constant differ from a pointer variable? Give an example of each from the program in this lab.

   8.2 How does mystr differ from mystr[2]? What data type is each variable?

   8.3 What is wrong with this assignment?: p_mystr = mystr[2];

   8.4 Why is this assignment correct?: *p_mystr= mystr[2];

   8.5 Define the following terms: source code, debug, pointer, array element, string constant.

Unit 5

UNIT 5: PROG-2000 Introduction to Microcontrollers

Z8 Hardware interfacing and Delay loops

Unit Objectives:

The student will be able to:
1. Correctly wire an interface to the Z8 Microcontroller evaluation board.
2. Draw a flowchart to describe the operation of a program.
3. Test program conditions using relational and logical operators.
4. Use variables to count program events.
5. Correctly write delay loops using a "for" statement.
6. Distinguish which data type is appropriate to use depending on the variable size.
7. Measure the execution time of a programming event using an oscilloscope.

In this unit student should do the following:

1. Re-read chapters 10 and 11 of "C by Example" on loops. Don't forget to make notes on what you have read.
2. Read the handout: "Structured Programming and Flowcharts".
3. Describe on paper, the difference between: an "if" statement, a "while" loop, a "do..while" loop, and a "for" loop. Put this assignment in your Lab Manual (your duo-tang folder) along with the rest of Lab 5. ***This will be part of your mark.***
4. Do Lab 5. The first program, 5.1 (lab5_1.c) is used to test your hardware and wiring. Study how it works and try it out. ***Troubleshoot*** any hardware connection errors. Next, study the requirements for program assignment 5.3. Then, draw a ***flowchart*** that describes ***how*** programming assignment 5.3 is supposed to work.
5. Write program 5.3 (lab5_3.c) based on your flowchart. Put this lab, along with your flowchart, in your Lab Manual. Demonstrate your running program to your instructor.

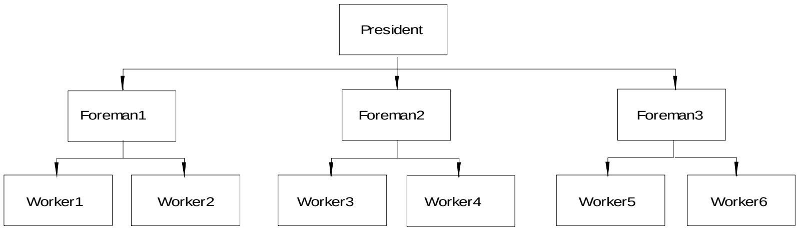
Note: You will be using the using the bargraph LED's and DIP switches for subsequent labs so leave those components connected, until instructed otherwise.

UNIT 5: PROG-2000 Introduction to Microcontrollers

Notes: Structured Programming and Flowcharts

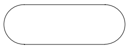
1. How do you get started, writing a program? The first thing that you have to do is specify the problem that you want to solve. We'll start with a simple problem that everyone can understand. The next step would be to design a flow chart to represent what you want your program to do. In this case we'll analyze a flowchart that represents the tasks that are involved in describing this problem. A flow chart is a graphical way of representing those tasks.
2. In flow charting, each step in a problem has its own block. By connecting the blocks with arrows we show the order or sequence the of the steps or tasks, that need to be done. Sometimes we have a choice to make that affects which direction or which operation we will perform next. For that reason flow charts are sometimes referred to as “decision trees”. Flow charts can be used to describe just about any task we need to perform.
3. Here's your job. You need to explain ***how*** to make a telephone call to someone who has never used a phone before. A ***flow chart*** is the tool that you will use to provide this explanation. I've already written the flow chart for you, so we will analyze this flow chart to make sure, that it will actually teach someone how to make a telephone call. The procedure used to make this flow chart is the same as we would use to design a flow chart for a program. After studying this example flow chart, you should be ready to draw your own.
4. When designing a program we will use a methodology called “***Top Down Design***”. This technique allows you to concentrate on the big problems and leave the out the details, until the structure of your program is defined. Only then will you tackle each of the smaller problems. Each high level task (a big problem) is one ***step*** in producing the logic that will allow you to complete the job (a program that works). Only ***after*** this planning stage (the flow chart) is completed, should you begin to write your actual C program.
5. Another concept included in “Top Down Design” is that of ***hierarchy.*** An example of a hierarchal structure is the organizational chart of a company. For example, a construction company has a president. He is in charge of choosing which projects he would like to build. He may choose 3 projects. So he hires 3 foremen to oversee the work on each of the projects. However, the foremen don't do the actual work themselves. They are in charge of hiring the tradesman and assigning them individual tasks. Let's look at a different situation. The president might want to send a message to someone. He tells the foreman ***when*** the message must be delivered. The foreman is the one who must decide ***how*** it will be delivered. If it is not required for several weeks he may tell the worker to mail it. If it must be delivered this afternoon, he would tell his worker to fax it. Do you notice the division of labor? The president and the foreman each make decisions, but at a different level. The worker still does the work. Top down design works the same way. The same hierarchal concept can also be applied to program modules. In C we call these program modules “***functions***”. C functions often have a similar division of labor depending on their hierarchy. Each function should be a building block that only does one job, even if that job requires many steps. The lowest level functions often handle the I/O chores and do the work of interfacing to the outside world.

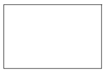
Example of Hierarchy: A company Organizational Chart

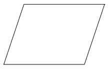


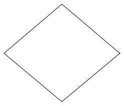
6. Let's look at an example of how hierarchy is implemented, in the C language. The main() function can be thought of as the president. In main() you will find a series of function calls that make up the general structure of the program. You might also find “decision” (if or if..else) statements that do ***high level*** decision making. These functions in turn, might call other functions that look after the details of a problem (such as such as mailing or faxing).

7. Below is a list of different symbols we will use for our flow chart diagram.

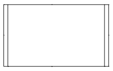

Start or End of Program or Module


Process (an operation or a task)

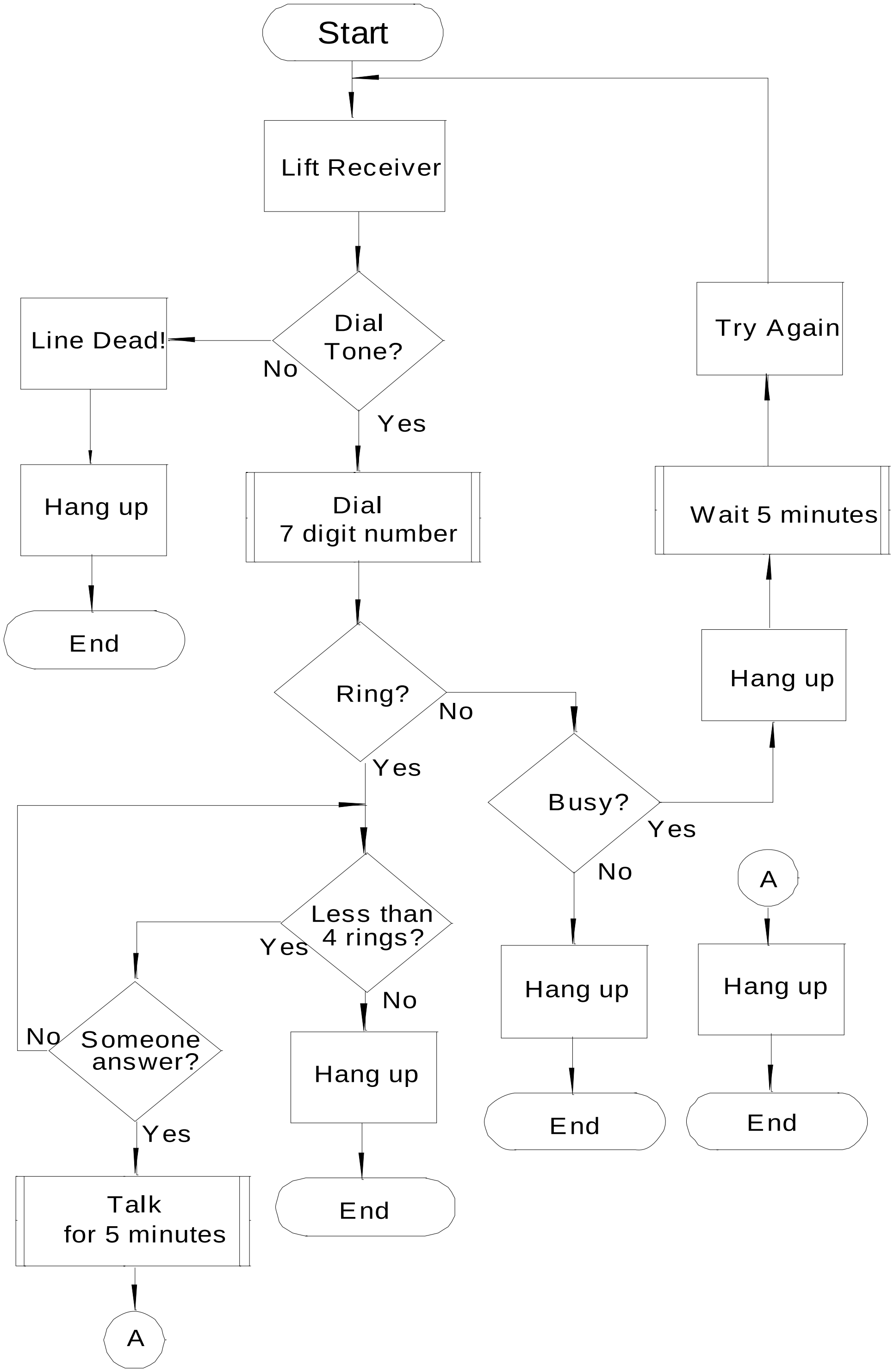

I/P or O/P


Decision

Connector


Function or Module

8. Next we’ll see an example of how a flow chart can represent the procedure to call someone on a telephone. Since you all understand how to use a phone, you can concentrate on ***how*** the steps and decisions are implemented. If you can specify what you want to accomplish, you should be able to implement a flow chart to represent it. Follow through the steps of this flowchart. Note how multiple step operations are implemented in ***“functions”***. Examine how the ***decisions*** affect the program flow. Would you be able to make a phone call after studying this flow chart?


Start
Lift Receiver
Dial Tone?
No
Line Dead!
Yes
Hang up
End
Dial 7 digit number
Ring?
No
Yes
Busy?
Yes
No
Less than 4 rings?
Yes
No
Someone answer?
No
Yes
Talk for 5 minutes
A
Hang up
End
Hang up
End
Hang up
Wait 5 minutes
Try Again
A
Hang up
End

9. When you start designing ***your*** flow chart, think through the steps involved in ***one*** aspect of the problem. Put something down on paper, even if you don't have everything figured out yet. Remember you are not concerned about the details of the problem yet, only the overall operation. Next, look at what you put down:

- Is there another point that you can add in?
- Where will it fit?
- Is there some step that you missed?
- Does the ***logic*** make sense?

10. Keep ***adding*** to your flowchart. Each time you add something, ***stop and review*** what you put down. Finally ***correct*** your mistakes (logical errors) and add, review and correct some more. Eventually you come to the point where you have completed the structure of your program. As you gain experience, you will be faster and more efficient at doing this.

11. Now that you have a framework for your program, you can start looking at the details you need to consider. It's a good idea to follow the same steps outlined in 9 and 10 when implementing the details. It's like another small program. This code will probably be placed in a ***function***.

12. ***Now*** is the time to start! When you get a programming assignment, follow these steps and write a flow chart. As you practice you will probably start to enjoy the mental challenge of organizing the steps of your program into a flowchart. This sort of puzzle is more fun than a video game (plus it's cheaper and doesn't ruin your eyes).

PROG-2000: Intro to Micros

The following parts are loaned to the student for use in this course. The student agrees to return the parts at the end of the semester.

Parts List:

1. 1- 10K ohm SIP resistor pack. ____
2. 1- 390 ohm SIP resistor pack. ____
3. 1- 100 ohm DIP resistor pack. ____
4. 1- 10 element LED display. ____
5. 1- 10 position DIP switch. ____
6. 1- 18 inch, 40 conductor IDE cable. ____

Received ______________________________ Date______________________

Returned ______________________________ Date______________________

1. 1- 16 character by 2 line LCD display. ____
2. 2- 18 inch, 40 conductor IDE cables. ____
3. 1- 10K ohm SIP resistor pack. ____
4. 1- 100 ohm DIP resistor pack. ____
5. 1- 12 key matrix keypad. ____

Received ______________________________ Date______________________

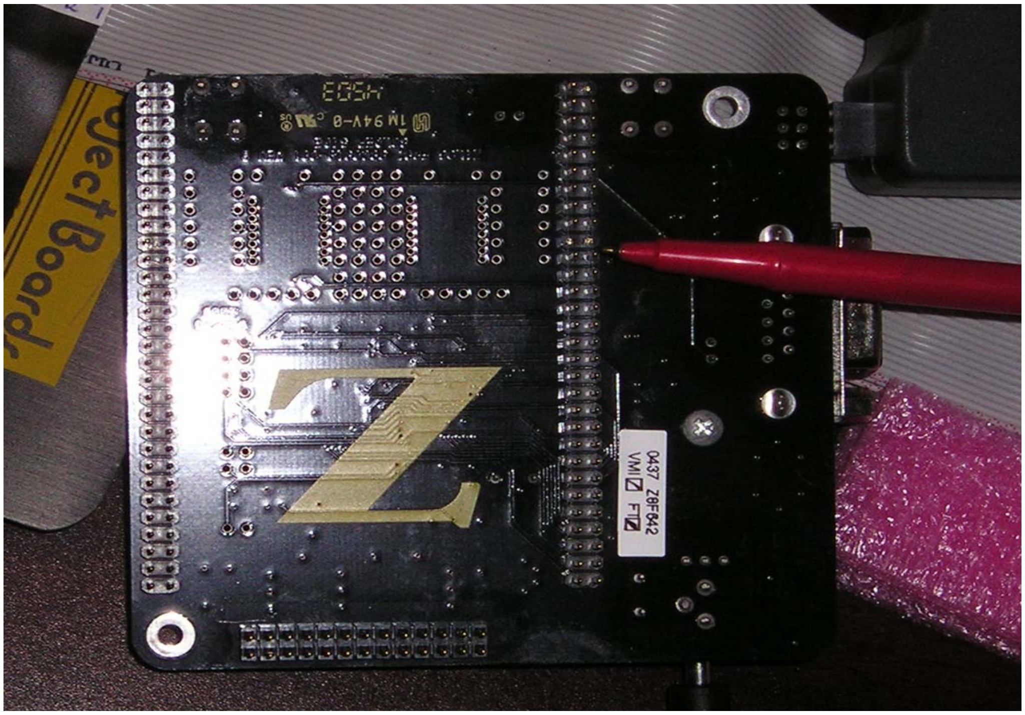
Returned ______________________________ Date______________________

UNIT 5: Modifying the Z8 Encore! Microcontroller kit for use as a Lab Tool

Remember to have your anti-static strap attached between you and ground when working with your kit! Wear your safety glasses!

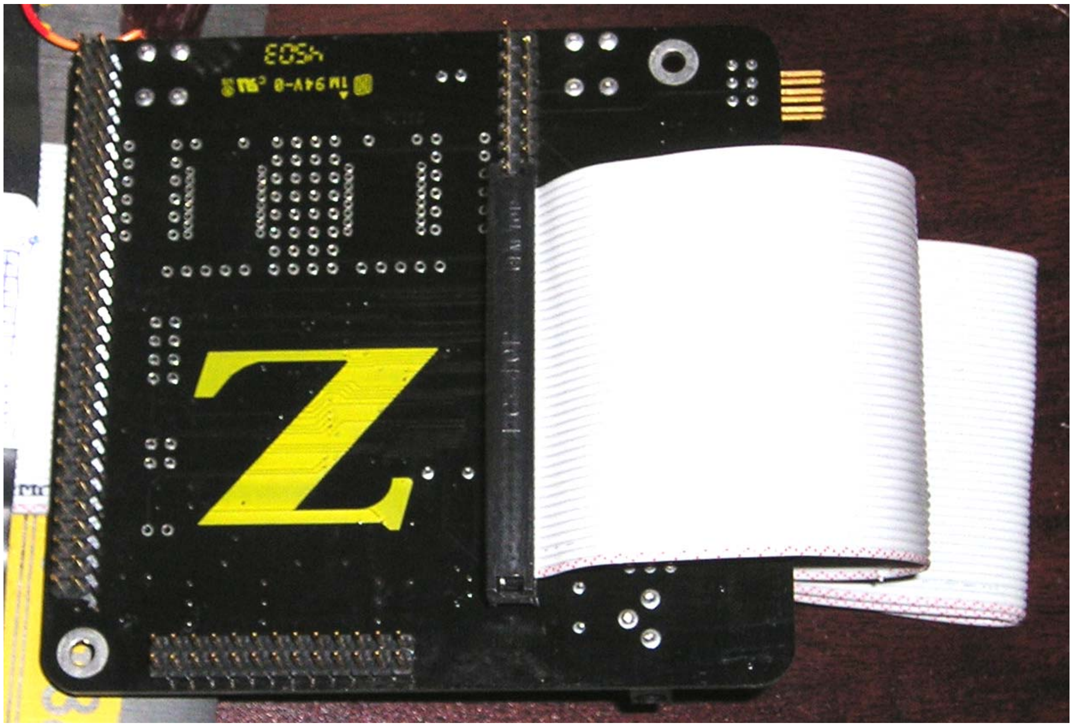
1. Locate JP2 from the top of the circuit board. Flip the board over to see the extended pins for JP2. You will carefully clip pins 41 and 42 from this bottom side using 4 inch side cutters. ***Before you do*** have someone verify that you have selected the correct pins on the correct connector. In Fig. 1 the tip of the red pen points just below pin 42. If you check your schematic, you will notice that these pins are not connected to any circuit.

Fig. 1



With those pins clipped, you will be able to connect a 40 pin IDE cable receptacle to the first 40 pins. Note the ***colored*** indicator on one side of the flat cabel. Connect ***this*** side of the connector to pins 1, 2 of JP2. The clipped pins allow the connector to fit over pins 1 to 40 of JP2. Fig. 2 shows the 40 pin IDE cable correctly connected to JP2.

Fig. 2



3. Next you will cut the pink, anti-static, packaging foam to allow the IDE cable to stay connected, with the foam used as a support base for the Z8 board. Carefully cut the foam using an x-acto knife, keeping your fingers out of harm's way. See Fig. 3 for a model. The thickness or height of the foam is about 11/16 of an inch.

Fig. 3

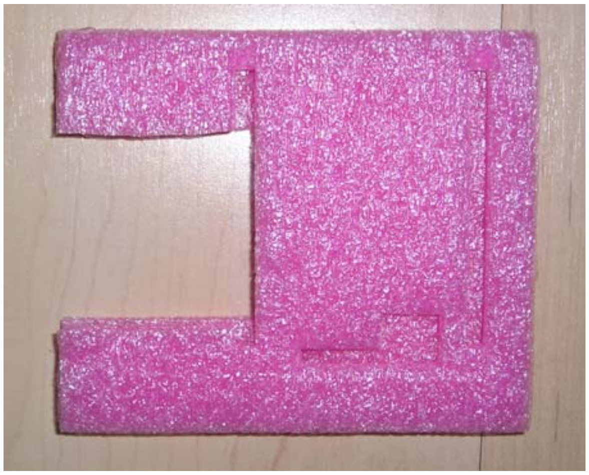
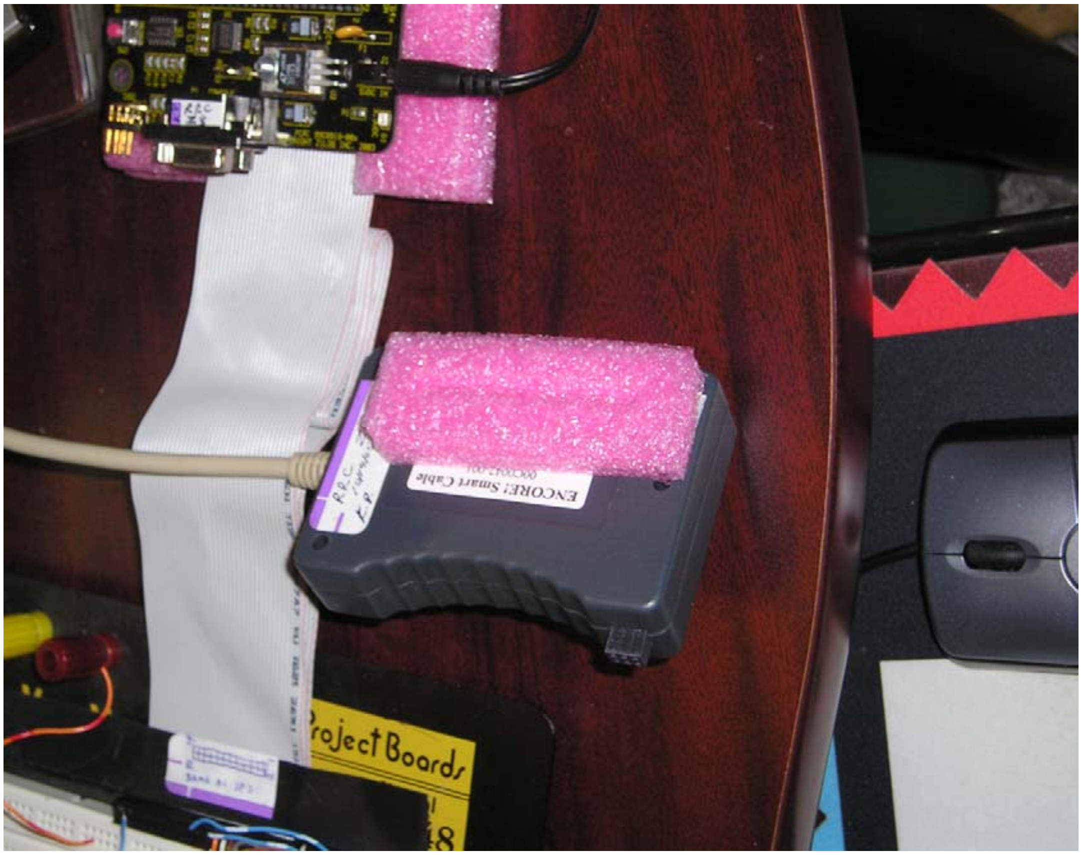
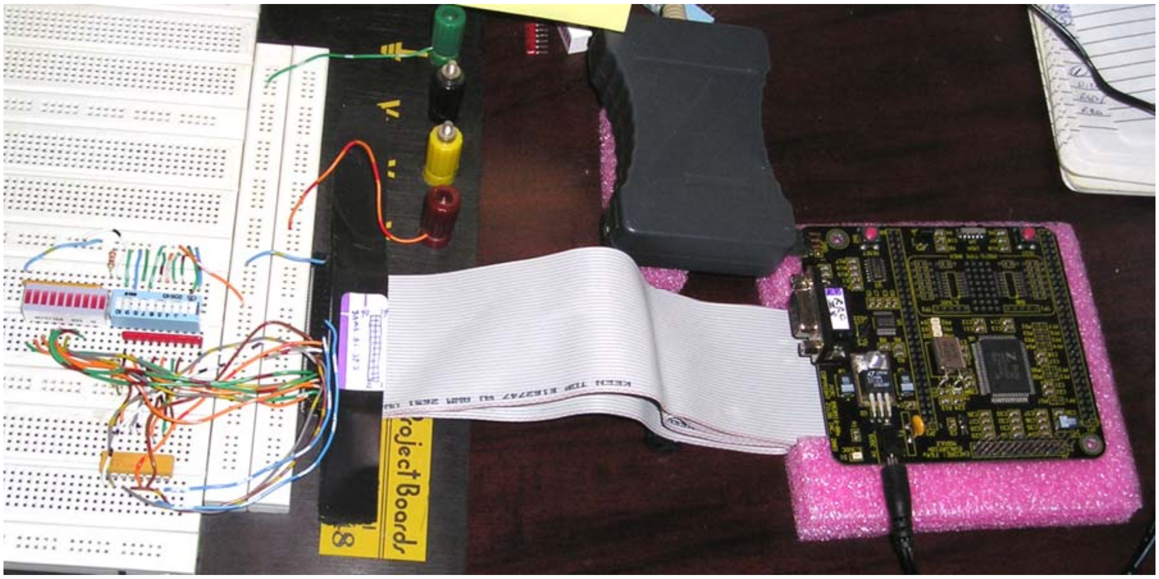


Fig. 4

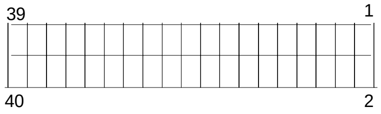


4. Finally, here is what you should end up with. You will notice that in Fig. 5, the other end of the IDE cable connector is next to the bread board, with it's sockets facing up. I have just used these sockets as connectors. You can use 22 gage wire to connect between it and your bread board. This is how you will interface between the Z8 ports and your own circuits. I used black electrical tape to keep it in place on the base of my breadboard. If you don't have a base you can glue (or screw) your breadboard to a piece of thin plywood, aluminum or plastic. It just needs to extend a couple of inches past the edge to provide room for the connector. The pinouts for the connector will be identical to JP2. The connector end next to the red edge indicates pins 1, 2 just as you see labeled on the JP2 silkscreen on the top of the circuit board.

Fig. 5



5. You will verify the pinout and integrity of the cable and connectors, by using a multimeter as a continuity tester. Get your partner to help you, and you do the same for them. First, set the meter on the 200 ohm range. Place one probe on the JP2 connector pin (from the top of the circuit board, start with pin 1) and the other probe to a 22 gage wire inserted in pin 1 of the socket end of the IDE connector. There should be a short circuit between JP2 pin 1 and pin 1 of the IDE connector by your bread board. Do the same for the other 39 connections. I actually had two open connections on the first cable I tried. Next, draw a pinout legend on an adhesive backed label. You can then put this legend close to your connector, for easy reference. Here is an example legend:



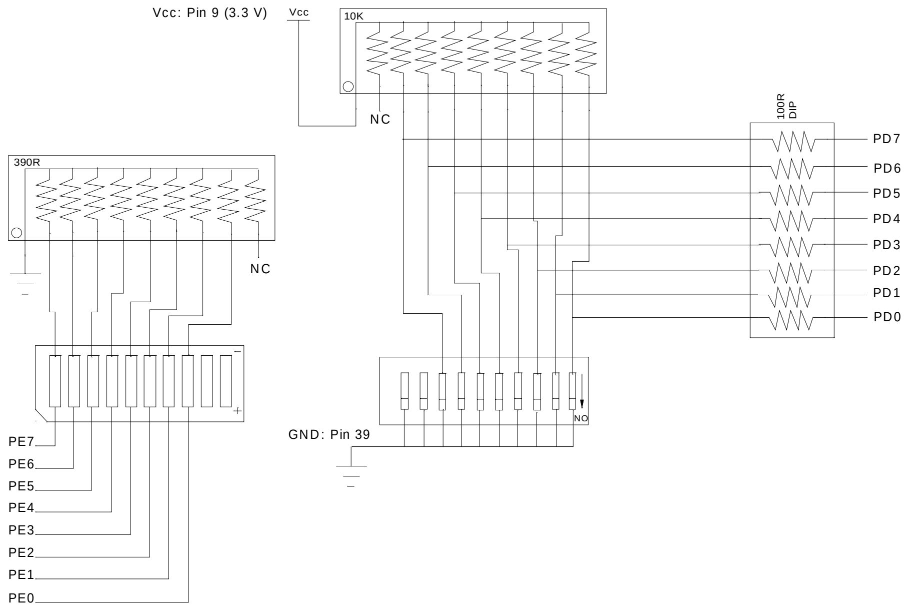
There! You should be all set to interface to the Z8.

UNIT 5: PROG-2000 Introduction to Microcontrollers

Lab 5.1:

1. Check off the parts on the supplied parts list, that you received from your instructor. Sign the sheet, indicating that you received the parts and ***return*** the sheet to your instructor.
2. ***Carefully*** follow the steps provided in the handout received previously, "Modifying the Z8 Encore! Microcontroller Kit for use as a Lab Tool". Clip pins 41 and 42.
3. Build the following circuit up on your breadboard. Double check your wiring. Make sure to make the proper connections to the IDE cable for the required ports. Verify the VCC and ground connections, by measuring the supply voltage (3.3 Volts) on the breadboard.
4. Note that there are 2 different S.I.P packages: 10K ohm and 390 ohm. The common on the 10K is connected to VCC but the common on the 390 is connected to GND. The 100 ohm D.I.P. Package has independent resistors connected to opposite pins. The purpose of these is to protect your Z8 ports if the connected pins are accidentally programmed as O/Ps, instead of I/Ps.
5. Also note that I have drawn the D.I.P. Switch pack with the ON position being down. If you wire it this way, then UP is logic 1 and DOWN (On) is logic 0. This might be less confusing when entering binary data. The LED display anodes are on the same side as the small beveled corner and the corresponding cathode is the pin on the opposite side.

Bread board interface circuit



6. Get out your Z8 Kit schematic. Look at the connections for JP2. Mark the IDE cable pin numbers of the corresponding port lines for ports D and E on your lab 4 “Breadboard Interface Schematic” next to the port lines (ie. PD7 – Pin 29). Using 22 gage wire, connect the port lines to the correct components. Don't forget VCC and GND connections. ***Verify*** the IDE cable wiring, to JP2 pinouts, as per the handout “Modifying the Z8 Encore! Microcontroller kit for use as a Lab Tool”.
7. Make a ***project*** from the following C program. This program sets the LEDs to the logic level of the corresponding DIP switch. Build, reset and execute it. If it runs correctly, it will verify that your hardware is hooked up correctly.

```
// Add your own Information Header. Call this program Lab5_1.c
// This program uses Port D and E for GPIO.
// Port D is initialized for I/P only, Port E for O/P only
// and the Alt Function is disabled

        #include <ez8.h> // GPI/O
   void main()
 {
                // variable declaration
                int val;

                //initialize 8 bit port E0(pin8)-E7(pin0)
                PEADDR = 0x02; // alt function
                PECTL = 0x00; // no alt function
                PEADDR = 0x01; // data dir
                PECTL = 0x00; // set all 8 bits to O/P
                PEADDR = 0x03; // O/P control function
                PECTL = 0x00; // set to push pull rather than open drain
                PEADDR = 0x00; // prevents inadvertent changes to sub registers

                //initialize 8 bit port E0(pin8)-E7(pin0)
                PDADDR = 0x02; // alt function
                PDCTL = 0x00; // no alt function
                PDADDR = 0x01; // data dir
                PDCTL = 0xFF; // set all 8 bits to I/P
                PDADDR = 0x00; // prevents inadvertent changes to sub registers

                // Read Port D
                while(1){
                        val = PDIN;
                        PEOUT = val; // display I/P from port D on Port E
                } //end while
  return; // done!
 }
```

UNIT 5: PROG-2000 Introduction to Microcontrollers and C Programming

Programming Assignment lab 5.2

1. After verifying that lab5_1.c works properly on your Z8 development kit, create a new project that will display a binary count on the 8 LEDs.
2. Make a project from “**lab5_2.c**” below. This program will increment “count” each time the loop is repeated and display the binary value of “count” on the LEDs:

```
//#include files
void main(void)
{
   //variable declarations first
         unsigned int delay;
         unsigned char count=0;
   // initialize PORT E here!

         while(1){
                  PEOUT=count;
                  count++;
         }//end while
}// end of main
```

3. If you single step through the program, it will operate as expected. However, when the program runs at full speed, it is much too fast to follow. We need to slow things down. The required time delay can be provided by a “for” loop that is used as a “count down” timer. Add in the following code, after “count++”:

```
for(delay=0xffff;delay>0;delay--)
 ;  //do nothing if the condition is TRUE
```

   How does it work? The variable “delay” is initialized to all 1's (0xffff) which is 65535 in decimal. Next, the test condition is evaluated. The value in “delay” is compared to 0. If “delay> 0” the condition is TRUE. Since the condition is TRUE the next statement (the ';') is executed. The “;” just tells the CPU to do nothing for one instruction period. After that, “delay” is decremented and the cycle repeats until the test condition is FALSE. If the condition is FALSE, the “for” loop will terminate and the program will continue on to the next C statement.

4. Can you make the delay longer? Not if “delay” is declared as an ***int***. The largest number that can be stored in an int variable is 0xffff. This is 65535 times through the loop. If we want to loop longer, we need a bigger variable. If we declare “delay” as a “long int”, we can loop more than 4 billion times (a “long int” holds 32 bits). Try it out with a delay value of 200,000 and see how it works.

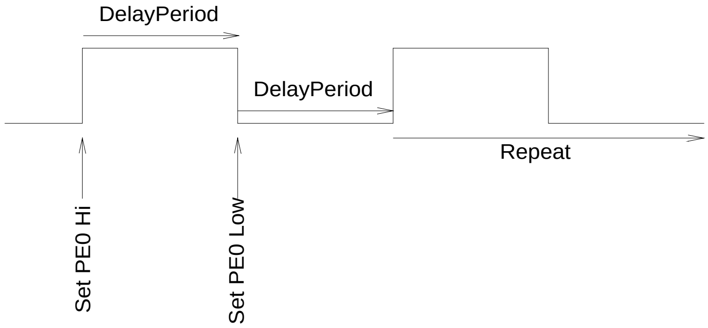
Continued on next page.

5. Can you measure the actual ***delay time*** of our delay loop? Yes you can! Modify lab5_2.c as follows:

```
//#include files
void main(void)
{
   //variable declarations first
   long int delay;//for big delays
   //add and initialize PORT E here!
         while(1){
                     PEOUT=0x01; // set PE0 (the lsb of PORT E) to logic 1
                           for(delay=1000;delay>0;delay--) // loop for a 1000 times
                           ;//do nothing
                                    PEOUT=0x00;// clear PE0 to logic 0
                           for(delay=1000;delay>0;delay--)
                           ;//do nothing again
         }//repeat forever
   }// end of main
```

Monitor port line PE0 with an oscilloscope probe. You should see a fixed frequency square wave.

- PE0 goes Hi.
- Delay
- PE0 goes Low
- Delay and Repeat



The period (T), of the square wave is the sum of the Hi time and the Low time of PE0. Measure either the Hi time ***or*** the Low time of the waveform, using an Oscilloscope (ask your instructor for a probe). Divide this time by 1000 (the number of times the loop executes). The result is the time it takes to execute the "for" loop, once. Now, if you want to calculate the *number of times through the loop* for a 1 second delay, divide 1 second by the time it takes to execute the loop once. Here are the steps in point form:

- Time for one loop (**LoopPeriod**) = Hi time / 1000 (because we assigned 1000 to "delay")
- **LoopCount** (to delay 1 second) = 1 sec / **LoopPeriod**
- Assign **LoopCount** to "delay" in the "for" loop. For example:
  for(delay=LoopCount;delay>0;delay--)
  ;

6. **Programming Assignment Lab 5.3:** Modify program lab5_1.c (call this new program, lab5_3.c) to accomplish the following:

- The 8 LED display will show the binary count of a variable starting from 0x00. Each count will be displayed for **2 seconds**, then will increment by 1. Use an char variable for count. That way if the count value exceeds 0xFF, it will roll over and continue again from 0x00. The value of count will display on the LEDs unless ***both*** of these conditions are met: an ASCII 's' is present on the DIP switches (note which end of the switch is the msb) ***and*** the count variable is greater than 0x14. If this condition occurs, O/P the ASCII value of 's' to the LED's. If the condition is no longer true then resume counting on the display. Use an "if" statement and ***relationa***l operators, to test the 8 bit value of the DIP switch and the value in the count variable. Use logical operators to make sure both conditions are met.
- Add a "for" statement implemented as a count down timer. It will be used for a ***time delay*** with a period of 2 seconds, to allow you to easily read the count on the LED display. Use the information in lab5_2 to implement the correct delay time.
- **Draw a Flowchart**. Before you begin writing any code, you need an action plan. Preparing a ***flow chart*** is an excellent method of designing your program ***before*** you write your code. As a reminder, review your handout on "Structured Programming". Flowcharting can help you to understand ***how*** to accomplish your programming task and sort out any logical errors ***before*** you start coding. ***Include your flow chart*** as part of your programming assignment.

7. ***Demonstrate your program to your instructor.*** Don't forget to properly label your **flow chart**, your commented source code (lab5_2.c and lab5_3.c) as well as Assignment 5.2 (from Unit 5 Objectives/Assignments page) and put them all in your lab book as part of Unit 5.

Unit 6

UNIT 6: PROG-2000 Introduction to Microcontrollers

C Functions

Unit Objectives:

The student will be able to:

1. Correctly interpret and write functions prototypes.
2. Correctly identify parameter data types in function headers.
3. Write functions that accept one or more parameters.
4. Write functions that return a parameter.
5. Correctly make a function call from another function.
6. Write function code based on a supplied flowchart.
7. Debounce mechanical switches using software.

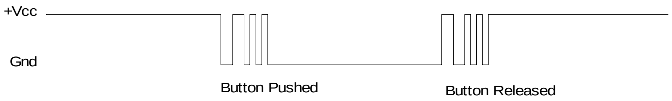
In this unit student should do the following:

1. How do you write ***functions*** in C? Read chapters 13, 14, 15 and 16. Function prototypes are described, starting on p.285 of chapter 16.
2. Do lab 6. Demonstrate your program's operation to your instructor.
3. Put your commented source code in your Lab Manual and show your instructor.

UNIT 6: PROG-2000 Introduction to Microcontrollers

Lab 6

1. When you were running lab3_1.c did you notice that pushing the “test” button produced erratic results when you executed your program at full speed? In contrast, it worked fine when you ***single stepped*** through the program code. Why was that? This was caused by the switch bounce of the test push button switch (refer to the TEST push button switch on your Z8 Development board schematic). You can expect about 600 microsecond of bounce from a typical pushbutton switch. Below is an exaggerated illustration of the problem.



   One way to eliminate the effect of switch bounce is with software delays. PC0 is connected to the test switch O/P and is a logic '1' except when the switch is pushed (closed). The procedure would be as follows:
   - check for a logic 0 (button pushed),
   - delay 2 milliseconds (wait until bouncing stops), then test again that it is still 0.
   - check for a logic 1 (button released)
   - delay and check again for a logic 1 (wait until bouncing stops)

   Your task is to write a function that will perform this execute this sequence of steps. Put all the code in a function with the following prototype: **void button_pushed(void);**
2. The function “button_pushed()” calls other functions as described in the flowcharts on subsequent pages.
3. The delay function requires some additional explanation. Use the following prototype for this function: **void mydelay(unsigned int);**

   // pass an “int” to the delay function that will determine the delay time, in milliseconds. You will need a “nested” for loop for this function (See p.191 for an example). The inner loop will have a delay of 1 millisecond and the outer loop will repeat, based on the parameter sent to “mydelay( )”. The number (the parameter called “time”) you send to the mydelay(time) function will be the number of milliseconds that you will delay, because this is the number of times that you will execute the inner loop. When you need to perform a debounce delay, use the mydelay() function.
4. **Programming assignment Lab 6.1:** This programming assignment, will count each time the push button is pressed and display the number of presses on the LEDs, in binary. Naturally you will need to debounce the pushbutton switch before registering a count. Your program will call the button_pushed() function in order to detect button pushes. Model your C code on the pseudocode and flowcharts provided in the following pages. Use your new functions in your program and demonstrate it's operation to your instructor.

Flowcharts and Pseudocode

Lab 6 (call your program lab6_1.c)

//Required Function prototypes:

```
void init_ports(void);
unsigned int pushed_button(void);
unsigned int released_button(void);
void button_pushed(void);
void mydelay(unsigned int);
```

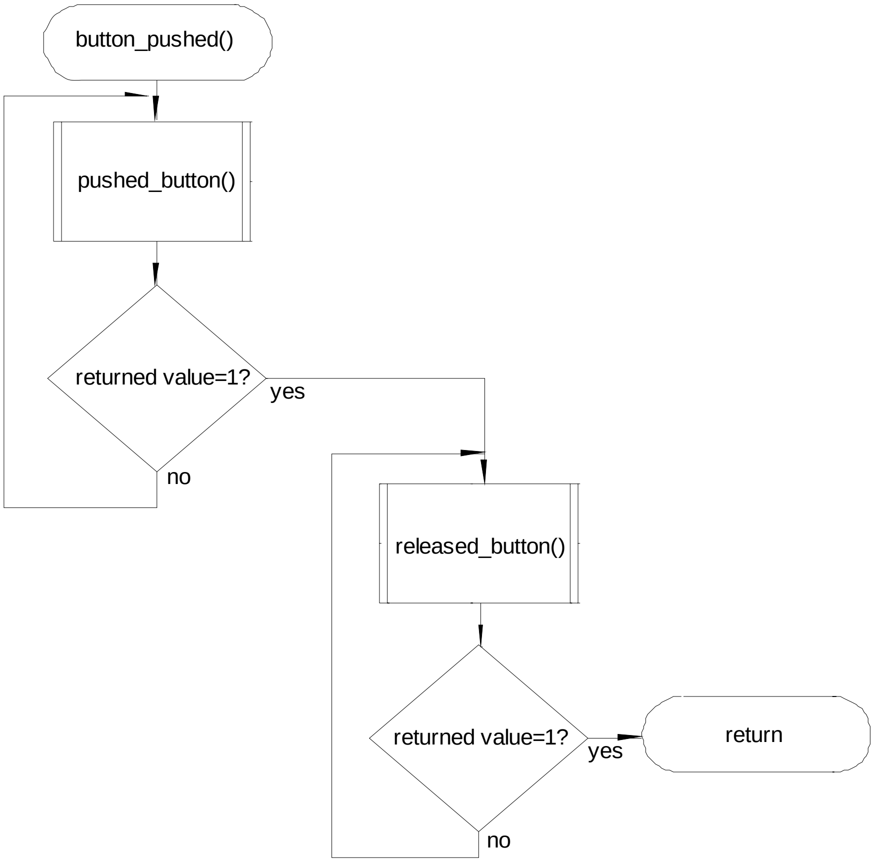
//pseudocode for **main()**

```
1.  Initialize Ports
2.  Initialize variables: count
3.  Loop forever
        a.  call button_pushed()
        b.  increment count
        c.  O/P count to LEDs
```

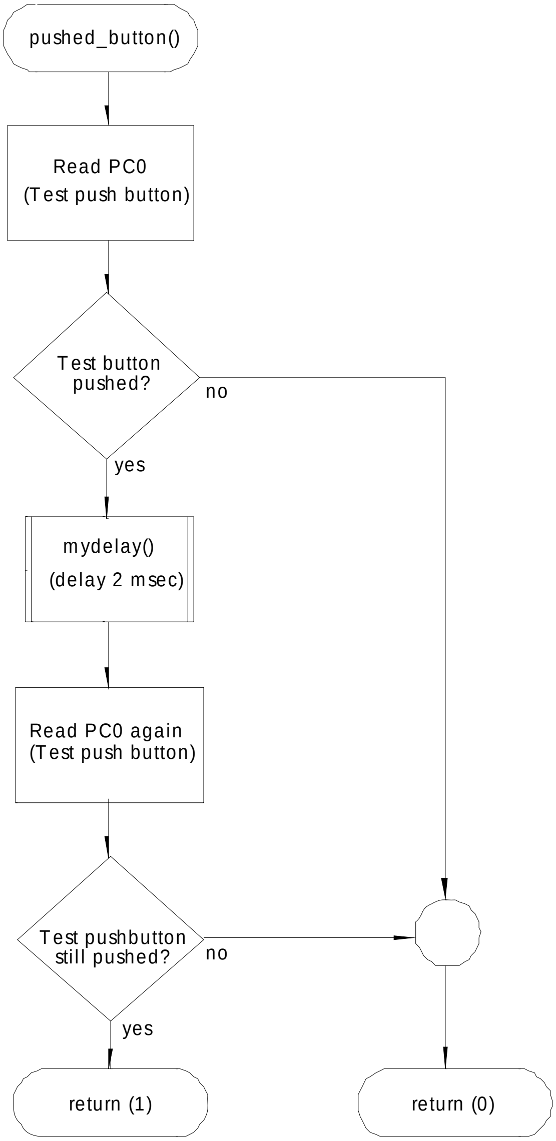
//pseudocode for **init_ports()** function

```
1. Initialize Port E for no alternate function, all bits O/P and push/pull
2. Initialize Port D for no alternate function, all bits I/P
3. Initialize Port C no alternate function, bit 0 as I/P (test switch)
```

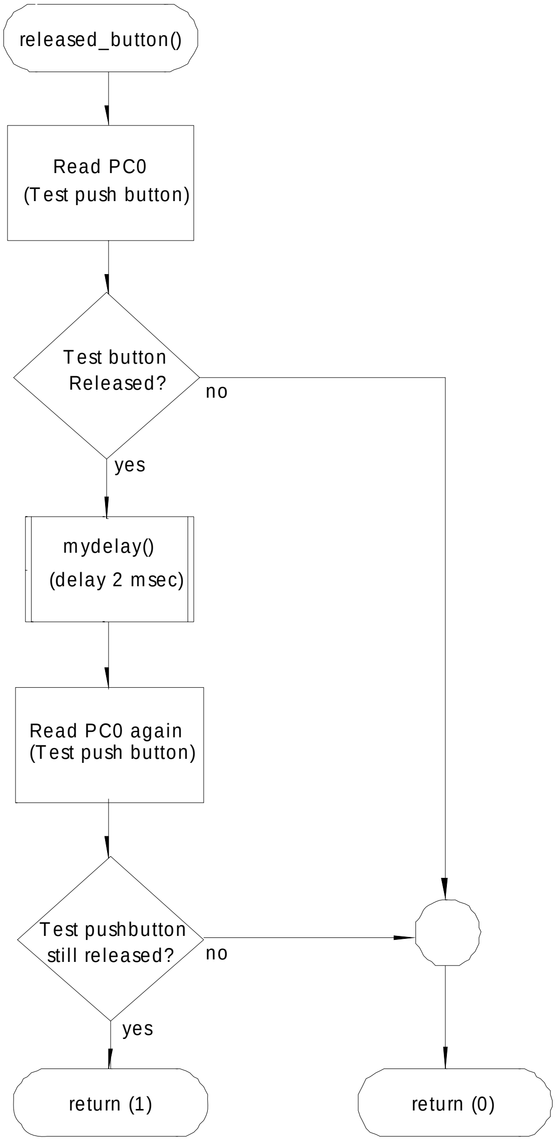
//Flowchart for **button_pushed()** function.
//pushed_button() and released_button() are called within **button_pushed()**.



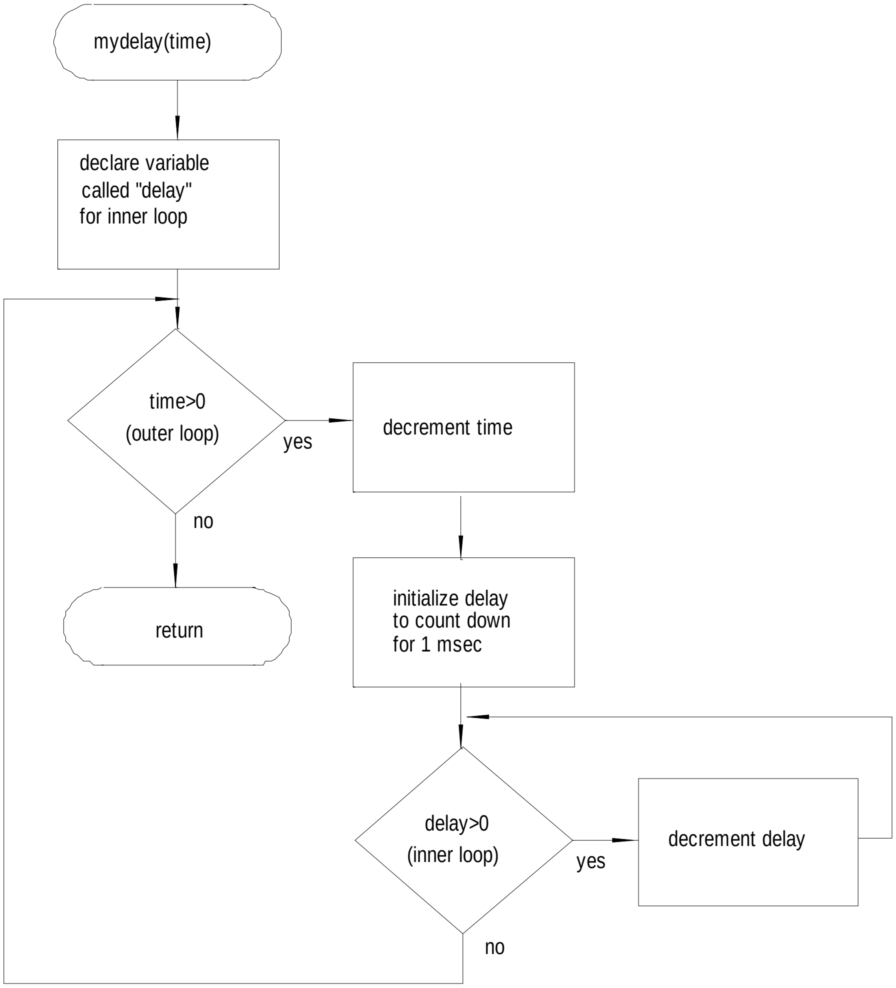
//Flowchart for **pushed_button()**, (called by button_pushed() function)



//Flowchart for **released_button()**, (called by button_pushed() function)



//Flowchart for **mydelay()**. It is called by pushed_button() and released_button().
//The mydelay() function, accepts a parameter called “time”, and is basically
//a “nested” **for** loop.



Unit 7

UNIT 7: PROG-2000 Introduction to Microcontrollers

Pulse Width Modulation

Unit Objectives:

The student will be able to...

1. Describe how a Pulse Width Modulated (PWM) waveform can be produced from a port line on the Z8.
2. Use the C "switch" statement in a C program.
3. Use an array as a lookup table, in a C program.
4. Interface a D.C. motor to a Z8 port line by using an isolation and drive circuit.
5. Describe the operation of a D.C. motor interface circuit.
6. Write a C program to control the speed of a motor, using PWM.

In this unit student should do the following:

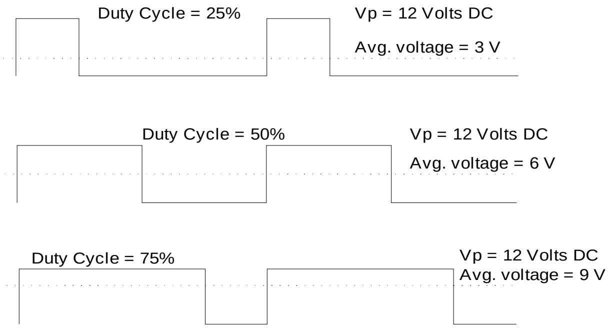
1. Read chapter 12, on the C "switch" statement.
2. Do lab 7. Write the required software to control the motor, following the specifications in the lab. Test the operation of your program while monitoring PG0, ***without*** connecting up the motor driver and motor. When it works correctly, wire up the motor drive circuit and motor, to your Z8 but be very careful! Remember, the power supply and ground connection for the motor is ***completely isolated*** from the Z8 power supply. The grounds are not common.
3. Describe at least 2 different applications that can make use of PWM.
4. Insert lab7_1.c source code into your lab manual and demonstrate the program's operation to your instructor.

UNIT 7: PROG-2000 Introduction to Microcontrollers

Lab 7

1. The objective of this lab is to control the speed of a DC motor using the Z8 microcontroller. The technique we will use to do this is called Pulse Width Modulation or PWM. Use lab7_1.c as the file name for your program.

2. As you recall, ***Duty Cycle*** is the ratio of the ***Pulse Width*** to the ***Period*** of a waveform (multiplied by 100 to give a % value). In PWM, the frequency of a pulse train is held constant, while the pulse width (and therefore the duty cycle) is changed. If the duty cycle increases, the average DC voltage (the voltage measured with a typical DMM) increases. Likewise if the duty cycle decreases the average DC voltage decreases. As long as the frequency of the pulse train is high enough, mechanical systems like a DC Motor can't tell that the voltage source is turning on or off. They can't respond to the fast changes and instead think a DC voltage (the average voltage) is being applied. A higher duty cycle will make the motor go faster, and visa versa. Refer to the diagrams below to make sure that you understand this concept. The frequency is 1 KHz for all three waveforms.



3. Write a function using the following prototype, to initialize all required Z8 port lines:

void init_ports(void);

Use port line **PG0** (pin 34) as the PWM O/P, and port lines **PD7 to PD0** for the DIP switch I/Ps (DIP switch S1 To PD0, S2 to PD1 etc.).

4. Design a ***flow chart*** for lab7_1.c, according to the following specification: PG0 of the Z8 will O/P a constant frequency pulse train. The duty cycle of the pulse train can vary, depending on the state of the DIP switches. Only switches S1, S2 and S3 are used. All other switches will be ignored. If S1 (connected to PD0) is logic 1 while S2 and S3 are logic 0 (0x01 if reading an 8 bit value from port D), then a duty cycle of 30% will be selected. If S1 and S2 are logic 1 (0x03), then a duty cycle of 50% will be selected. Only if S1 S2 and S3 are all logic 1 (0x07), will an 80% duty cycle be selected. ***All other switch combinations*** will select a 15% duty cycle. Your program is required to use a "switch..case" statement to determine which duty cycle

to select. After verifying that the logic of your flowchart makes sense, write the code for lab7_1.c. Write and test each part of your program separately, do not put it all together at once. Remember, start simple and add complexity.

5. The duty cycle will be determined by the use of a delay() function (prototype: **void mydelay(unsigned int);**). Use a single “for” loop in your delay function, with a delay time of 1 microsecond per parameter value. If a port line is held high for a longer period of time, the duty cycle increases. Naturally, if the High time is longer (higher duty cycle), then the Low time must be ***shorter*** in order to keep the period of the pulse train the same. Think about ***how*** this can be accomplished. The pulse train frequency should be 1000 Hz (T=1 msec). Use an array of unsigned int values to store the 4 duty cycle options that are allowed. That way you can select an array index that is unrelated to the actual delay time that is sent to the delay() function. This is an example of a ***lookup table***.

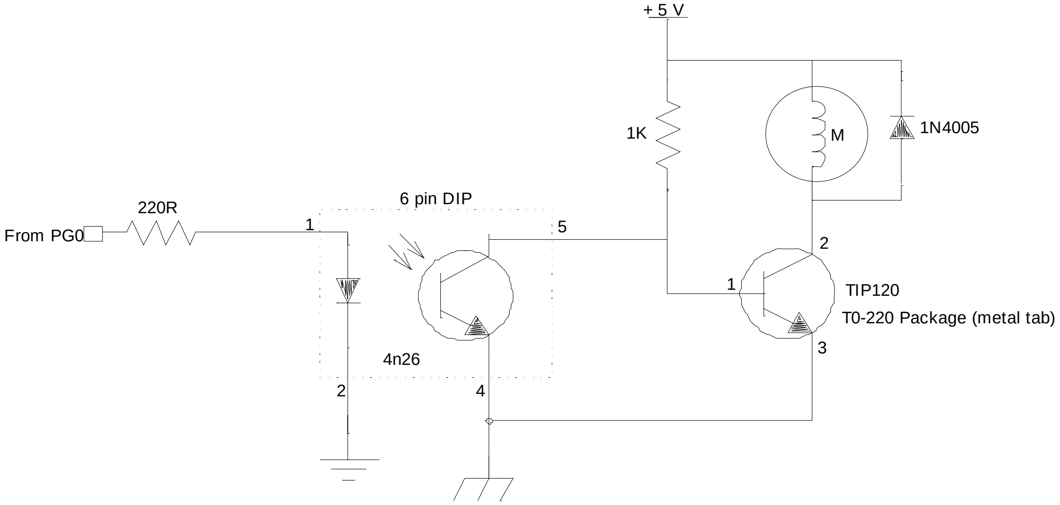
```
//simple lookup table example
unsigned int hi_time[4] = {300, 500, 800, 150};
//relative time values: 1 cycle is 1000 time units
//index 0 provides a 25% (250/1000) ON time
//index 1 provides a 50% ON time and so on
```

6. Once you have monitored the PWM waveform O/P on PG0 with an oscilloscope and your program works as expected, it's time to build your interface circuit. ***Build the circuit using the schematic on the next page*** with the parts supplied by your instructor. ***Return the parts to your instructor at the end of the lab***. You must be very careful that you hook things up correctly, in order to prevent damage to your Z8 development board. Recheck your wiring before applying power.

7. These ***components may be new*** to you. ***Lookup and read the component data sheets*** from the internet. The following is a brief description of the circuit. The 4n26 (6 pin DIP) opto-isolator is used to provide electrical isolation between the Z8 and your motor drive circuit. Inside the package is an LED and a photo transistor. Only light, not electrical current, links the Z8 to the photo transistor. PG0 is used to source current to the LED through a 220 ohm resistor. A logic 1 from PG0 will cause the LED to produce infra-red (IR) light inside the package. This IR light causes collector current to flow in the photo transistor (it has the same effect as applying base current). The photo transistor acts like an electronic switch. If IR light is applied, the collector and emitter becomes a closed switch. Therefore the collector to ground voltage is 0 volts. If the IR LED turns off, the collector to emitter voltage rises to VCC. This is because the collector to emitter becomes like an open switch. The collector of the photo transistor is connected to the base of the TIP120 power Darlington transistor. This transistor has a very high beta (>1000). It is also used as a switch. It is sinks the current through the motor when the photo transistor is cut off. It stops conducting when its base voltage drops to 0. The reverse biased diode is used to protect the transistor from high voltages generated by the motor when it is turned off abruptly. The Package pins are labeled on the schematic. Pin 1 on the TIP120 is the pin to the

left when looking at it straight on, with the tab underneath. The 5 Volt power supply is totally independent of the Z8. The grounds are not connected. Use the dedicated (non variable) 5 Volt output from the bench power supplies.

8. Once again, note how the interface circuit affects the polarity of the signal supplied from the Z8. When PG0 is high, is the motor on or off? Apply power and try it out. Can you change the motor speed as expected? Look at the various signals with an oscilloscope. Demonstrate the operation of your program and circuit, to your instructor.



End of PROG2000