**Lab 2: Designing a Microprocessor Based Test Instrument (Including Design Trade-offs)**

**EE475**

**02/09/2016**

**Electrical Engineering**

**Lingxiao Zhang**

**Xinyu Sui**

**Shuo Yin**

**TABLE OF CONTENTS**

Abstract 2

Introduction 2

Discussion of the Lab 3

Design Specification 3

Design Procedure 6

System Description 8

Software Implementation 15

Hardware Implementation 24

Testing  36

Test Plan 36

Test Specifications 37

Test Cases 38

Presentation, Discussion, and Analysis of Results 40

Analysis of errors 42

Failure Mode analysis 47

Final factory cost (BOM) 48

Final updated schedule 49

Summary & Conclusion 50

Appendices 50

Contribution Table 50

Signature 51

**Abstract**

This project mainly introduces how to use and operate the 18F452 PIC microprocessor, to utilize the knowledge of the SRAM from the previous lab, and use them to design a microprocessor based test instrument. The lab also introduces the full product developement life cycle while documenting, designing, building, and testing a simple instrument capable of measuring several kinds of data. For the software part, it mainly covers C language using the MPLAB, which is a integrated environment for the development of applications on the PIC microprocessor. For the hardware part, it mainly covers building a template circuit of the microprocessor based testing system, and the use of logic analyzer to analyze and debug the circuit design. By being familia with using these tools and understand the operations of the testing system, two phases needs to complete. In the first phase, we need to build a stand-alone version system that will incorporate all of the measurements’ capability specified in the lab spec and an EIA-232 interface to a local PC. In the second phase, we need to build a remotely controlled, network based system based on phase 1 with up to six other devices. We used the logic analyzer to debug our design. In general, we got our design works successfully and met all requirements in the end.

**Introduction**

The purpose of this lab is to design a microprocessor based test instrument. It mainly needs us to learn and work with some new tools and concepts as we design and incorprate a simple distributed, remotely controlled, network based instrument capable of measuring signal frequency, spectrum, time, and events. In the first phase, the project will focus on specifying, designing, building, and testing the basic instrument with all of the required measurement capabilties, such as signal spectrum, frequency, period, interval and events. We first need to get the communication between the PIC and the local PC. Then use the pickit-3 to program the code needed for measurements and communications on the PIC. The result of each measurement is to be displayed on an LCD. The timing constraints, the counting/timing portions of the design needs to be implemented in hardware outside of the PIC. The control signals coordinating all measurements shall be generated by the driver in the PIC. In the second phase, local PC based control, a local area network and a data stoage capability need to be added to the system/instrument. The local area network needs to be implemented as a SPI based network, which has a unit of master connected to the local PC with up to 6 slaves.

After reading and understanding the datasheet for an introductory discussion of the SPI, we need to develop a prototype, which generally supports the SPI interface, the LAN protocol and can be configured as any of 6 slaves. The application in phase 2 is an extension to the manually controlled system developed in phase 1 and there are detailed requiremtns provided in the lab specification. For the software part, we mainly used the uart to test the communications between the PIC and the local PC. For the hardware part, we mainly used the logic analyzer to test the behavior of the circuit design of our system based on skills we learnt from the previous project.

**Discussion of the Lab**

**Design Specification**

**Phase 1**

The phase 1 wants us to design a stand-alone version of the microprocessor based test instrument that will incorprate all of the measurement capability and an EIA-232 interface to a local PC. The testing instrument should have the capability of measuring following measurements with required measuring ranges, (some has two ranges of measuring, one for high range, the other for the low range):

1. Frequency in two ranges (digital)

High Range Upto 1.000 +/- 0.001 MHz

Low Range Upto 100.0 +/- 0.1 Hz

2. Period in two ranges (digital)

High Range Upto 10.00 +/- 0.01 ms

Low Range Upto 1.000 +/- 0.01 sec

3. Time Interval(digital)

High Range Upto 10.00 +/- 0.01 ms

Low Range Upto 1.000 +/- 0.01 sec

4. Spectrum(analog)

Range from 500-1000 Hz

5. Events

Range Upto 9999

Besides the measurements, SRAM and LCD are also specified to use for storing and display results of measurements. For the SRAM, the lab specification mentions that 16 most recent measurements must be stored in it. For the LCD, it’s used for displaying results of each measurement.

Finally, for the communication of the system between the PIC and the local PC, the local link is specified to have to support EIA-232 communication at 9600 bps with odd parity, and one stop bit.

**Phase 2**

The phase 2 wants us to incorporate an SPI interface and driver that will support a simple local area network(LAN) with up to six other devices. The final prototype to required to be the networked version of the instrument of the phase 1. The local area network is to be implemented as a SPI based network and should support one master and up to 6 slaves. The unit connected to the local PC shall be the master, and the remainder should be the slaves. The phase 2 is an application based on phase 1, which is an extension to the manually controlled instrument developed in phase 1. Below shows the figures of the application and the specifications of each layer of the whole design delivered:

**The Physical Layer**

The physical layer for our network will comprise 5 lines and their associated signals: transmitted data, received data, signal ground, and two control lines. Signaling levels will be those specified for RS 232 as described above.

**The Data Link Layer**

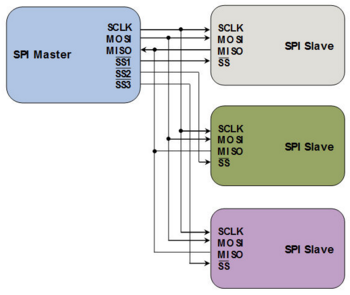
The data link layer will move characters, expressed in the EIA-232 format, from the host to a remote site and from a remote site to the host.

**The Protocol Laye**r

The Protocol layer will move commands and data from the host to a node at the remote site and move data from a node at the remote site to the host. Each node within such system is identified by a 3 bit address.

**The Application Layer**

The application layer ia based on the completions of physical layer, data link layer and he protocol layer and it must support the ability to command any of the instruments to make all measurements. The applicaion layer should meet the all specifications mentioned in the lab spec and the implements the SPI network as the figure shown below:



**Figure 1,** Design specification of the SPI network

1. For each instrument, the 16 most recent measurements must be stored in a RAM for display on demand. Such capability is typical in data logging applications.

2. The application must support the ability to retrieve the stored data from any instrument and display the data on the PC screen.

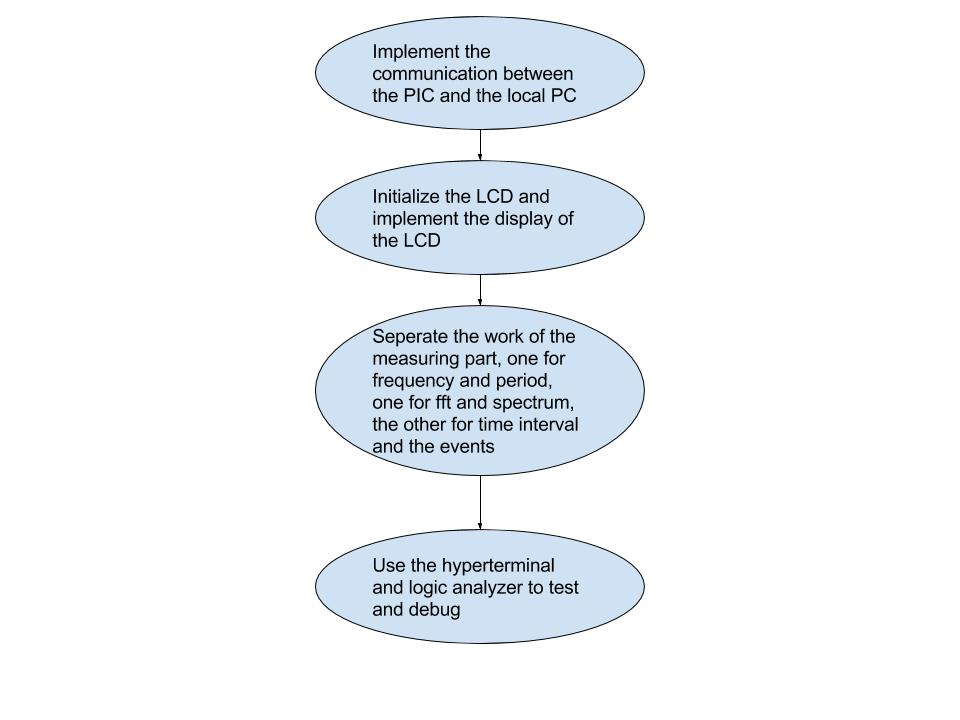
3. The local link has to support EIA-232 communication at 9600 bps with odd parity, and one stop bit.

4. The unit connected to the local PC shall be the SPI master in the LAN and the remainder of the instruments shall be slaves.

Finally, a full project report is required, including contents of updated deliverables and records of designing, building, and testing the complete instrument/system, to be conducted on the final report.

**Design Procedure**

**Phase 1**



**Figure 2,** Design Procedure of Phase 1

As the diagram of our design procedure shown above, below is our design procedure of phase 1,

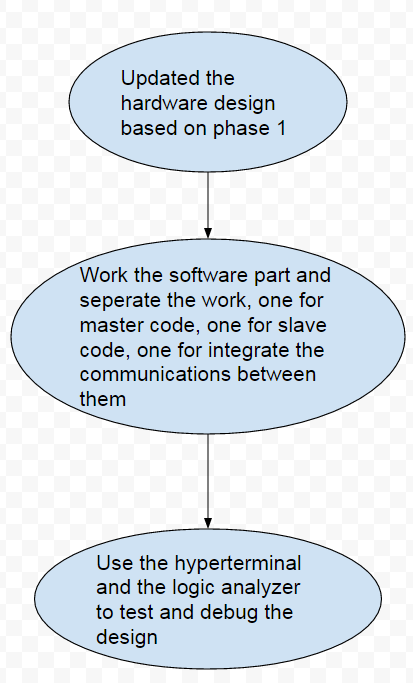
1. We first got the communication between the PIC and the local PC. We read the datasheet of the max232 chip, test and record the connection between the ports side connected to the PC and the wires side that connected to the max232 chip at first. Then we wired and made sure the connection between the PC, max232 and the PIC, and program the uart to the PIC and use the hyperterminal to test the commnications between the PIC and the local PC.

2. Then we worked on the LCD part, which is used for displaying the results of measurements, rather than working on the measurements part because we preferred to make certain there is no problem on the displaying results and then deal with the measuring part together. We first read the datasheet of the LCD, which includes the steps of initializing the LCD. Then we wrote the code of initializing the LCD follow the steps in the datasheet and type something on the hyperterminal and check and make sure the display is correct on the LCD.

3. We began to build and work on the measuring part of the instrument/system. We listed and prepared all chips will be used at first. Then we seperated the work and started work on both hardware and software part required. One person worked on the design and wiring part of the system and the function of counter read, which is used for control the count up of the counter to measure the frequency, one worked on the fft function for measuring the spectrum, the other worked on the timer function for measuring the timer interval.

4. We used the logic hyperterminal and the logic analyzer to test and solve the problem encountered after we finished building the circuit and wroting the code. Finally, we solved all the problems and finished phase 1.

**Phase 2**



**Figure 3,** Design Procedure of Phase 2

1. Because the phase 2 is an extension of the phase 1 and there isn’t a lot of things need to be added on the hardware, so we decided to have the updated the design of the hardware part first and then seperated the work into three parts based on our hardware design.

2. We began the software part of the phase 2 and seperated the work into three parts: master code, slave code and the communication between the master and the slave. We first went through the princples/algorithms of the SPI network from the lab spec and the search online, then we used what we’ve of master and slave from EE472 to finish our individual part.

3. Finally, after we finished all the code part, we began to test and integrate our design. We again used the hyperterminal and the logic analyzer to test the complete instrument/system we had built.

**System Description**

**System Input**

The system input for this lab is a signal which user may want to know about. This signal should be digital when the user need to know the frequency, period or events of the input signal, and when the user want to know the spectrum analysis of the input signlal, the user needs to make sure the input signal is analog.

In order to select correct mode of measuring, the user should also type in the HyperTerminal with the following commands to do specific measurement or slave command:

Master Commands

FH High Frequency

FL Low Frequency

PS Small Period

PL Large Period

IS Short Interval

IL Long Interval

SP Spectrum Analysis

SU Summary Report

Slave Commands

RA Low Frequency

RB High Frequency

RC Small Period

RD Large Period

RE Short Interval

RF Long Interval

RG Spectrum Anaysis

RH Reserved / Not Used

RI Most Recent 16 Measurements

RJ Summary Report

**System Output**

The system output is divided into two parts: the stats showing on the LCD, or internal vaules showing on the PC’s HyperTerminal. The stat on LCD will tell the user either frequency, period, event or the spectrum analysis of the input signal. Also, summary reports or most recent 16 measurements will also be shown to the user if corresponding commands are entered.

For developers and advanced users, they may also use the HyperTerminal software to get some more instant, but less explained internal values. These values, for example, frequency, updates at a relatively high frequency based on measurement system sampling rate, and each time 10 measurements are taken, an average will be calculated and reported to the user. But overall, these output are not that “visually appealing” to users as those on the LCD.

**Side Effects**

The system that is built in this lab does not produce significant side effects both to the input devices and the surrounding environmen while it is runningt.

The only problem of the circuit is that, since it contains many pieces of chips and wires, some static charge exists in the breadboards and sometimes cause gound pins of breadboards to appear wired in digital analyzer. Although after adding a few capacitors accross Vcc and GND this issue is not totally solved, the basic funtions of the system is not affected and we believe if commercial techniques of building the same circuit is applied, there shouldn’t exist similar noises.

Dispose of this device should be done with caution, as chips used in this device usually contains heavy metal elements which pollutes water bodies or soils. We highly recommend the users to dispose this device to certified electronic waste agent, if needed.

**Pseudo English description of algorithms, functions, or procedures**

1. User Interface

The user shall be able to select the following using the computer

Mode

Frequency, Period, Events, Spectrum

Range

Frequency, Period, Time Interval

High, Low

Events

Average of lastest 10 measurement

Number of measurements taken

Trigger Edge

Time interval, Frequency, Period and Events

Rising edge

Spectrum

Not triggered at any edge; starts upon user command

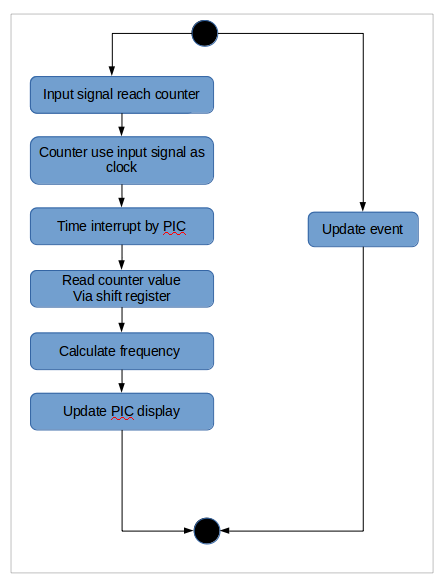
Reset

Power ON/OFF

The measurement results shall be presented on a LCD dispaly. Labels of each measurement will be shown so the user may understand the meaning of each data. The display shall be readable in direct sunlight from any angle.

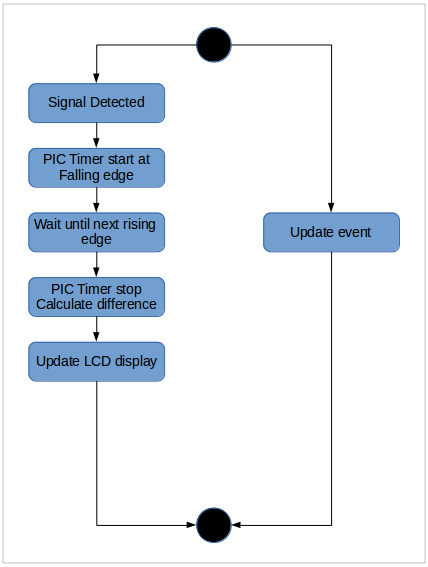
2. System Functional Description

The system is intended to make 4 different kinds of digital measurement in the time and frequency domains comprising frequency, period, time interval and events. The activities associated with the Measure Frequency mode are shown in the following diagram.



**Figure 4**, Flow chart of measuring frequency

The measurement of Time interval follows the same logic but it has minor differences.



**Figure 5.** flow chart of measuring period

The time and frequency measurements will be implemented to provide two user selectable ranges (High and Low frequency ranges) and user should select one of these to proceeed with the measurement.. For period, since we have masurement for frequency so it’s reasonable to just calculate the period by math (1 sec / frequency). For interval, two PIC clocks are used for the two cases of high / low frequency signal, and the user may also choose from high / low frequency measurement.

The event states are updated right after a new signal comes in. The user may choose to display or not to display a certain event on the LCD.

3. Precedure of measurements

Frequency Measurement:

During/while master PIC specified measuring time

always at positive edge of input signal

counter increment by one

Send counter value to PIC as frequency

Period Measurement:

Period = 1 / Frequency measurement

Spectrum Measurement:

While sampling with ADC at 4736 times per second

Send 256 sample data to FFT

Save result

Report result to PIC as spectrum measurement

Event Number Measurement:

At each calculation

Event amount ++

Report event number to PIC

4. Operating Specifications

The system shall operate in a standard commercial / industrial environment

Temperature Range 0-85C

Humidity up to 90% RH non-condensing

Power 120 – 240 VAC 50 Hz, 60 Hz, 400 Hz, 15 VDC

The system shall operate for a minimum of 8 hours on three or four fully charged AA batteries.

The system time base shall meet the following specifications

Temperature stability 0-50 C < 6 x 10-6

Aging Rate

90 day < 3 x 10-8

6 month < 6 x 10-7

1 year < 25 x 10-6

**Timing Constraints**

The general flow of working of the system is:

1. user trigger specified command on PC, and at this time, measurement system enters specified measuring mode

2. at each edge of input signal (for freq or period measurement), or at each ADC sampling time, either increment the counter or get a ADC data for FFT. After certain amout of sampling or counting, report the result to PIC.

3. After receving data, PIC will start to send it to LCD or PC terminal. For slave’s measurements, it will first send measurements to master and then master PIC will display the results to the users.

The most significant timing issues happens in the LCD subsystem as it requires certain amount of delays to initialze or, print something on the LCD. All the delays are specified in the LCD datasheet.

**Error Handling & Side effects**

For this system, errors may occur in measurements, and that is due to the delays in devices, signal noises, or mismanipulations.

To minimize the error of the measurements, there are serveral things can be done:

1. Use capacitors in circuits and apply circuit theories to reduce the output noises.

2. Remove possible environment harass, for example, electronics with big power consumption.

3. Be very careful during the manipulation, configuration process to reduce unnecessary errors.

4. Apply statistic knowleges on measurement data, use different methods to optimize data pool.

There is a 5% error of measurement expected at most for different measurements, and under actual industry conditions we expect the error to be even smaller.

**Software Implementation**

The overall flow chart of the system is show below.



**Figure 6,** Flow Chart of The System

After the master and slaves turned on, they do their initializations separately. Then the master waits for command from the user, or PC. This command sets the state of the master instrument. Then the master begin to do its tasks according to the command. This task may be done within the master instrument itself, or the master device may communicate with the slave instruments through SPI. In the latter case, that is, the slave receives a command form the master. Then the slave will do its tasks according to this command come from the master device. And this process goes infinitely. The tasks and other important details are discussed in detail below.

In this part, phase 1 and phase 2 of this project will be discussed together. Firstly, configurations are explained. Then, libraries/headers are discussed. Next, each function in the system is discussed. Specifically, important functions will be described in detail, while others may be omitted. In this segment, interrupt handlers and ISRs with their setups are also explained. Later, some small details are discussed. Finally, algorithms concerning communication and measurements will be discussed systematically.

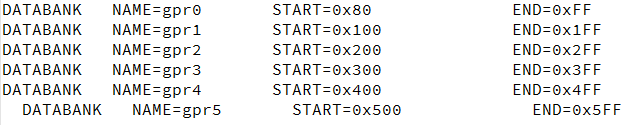
config.PNG

**Figure 7,** configurations before code body

Configurations before code body are shown in the figure above. WDT sets watch dog timer. Here, watch dog timer is turned off, since we believe for this system, if any crash happends, usually looking for the issue and restart manually is a smarter choice, rather than reset by watch dog timer. OSC configurates clock used for the PIC. In this case, ECIO means to use a External Clock while using the clock output as IO pin. LVP sets PGM (pin 38) on the PIC as a GPIO pin.

In this system, seven built-in headers/libraries are used: <p18f452.h>, <adc.h>, <usart.h>, <timers.h>, <stdio.h> and <spi.h>. <p18f452.h> is a header contains base address of all registers in PIC18f452. In our code, bitwise set up or assignment on the PIC are applied to variables predefined in this header file. <stdio.h> is included only for the *sprintf* function, which prints a constant string (char array) with predefined parameters (for example, %.5d) to a buffer. The functions of the rest of the headers are shown clearly by their names. Functions in these headers are usually in a consistent format. *OpenXxx* (*OpenADC*, for example) is used to open, or set up hardware such as ADC and UART. Note that, except *OpenXxx* functions, bitwise setups are also commonly used in our code for convenvience, and sometimes because many setups cannot be done by *OpenXxx* functions. *ReadXxx* functions are used to read values or receive data from various hardware, usually registers or serial port. *WriteXxx* functions are often used to write values to registers or transmit data to PC or other PICs (slaves). Data or values operated by *ReadXxx* and *WriteXxx* are char. For string operations, functions *PutsXxx*, *GetsXxx*, and *PutrsXxx* (for constant string) should be used. As their functions implies, *Put* functions are used to write or transmit, and *Get* functions are used to read or receive.

There are 26 functions in each of the master and the slave. Some of them are different while most are the same. Functions in common are discussed firstly. *delay(unsigned int t)* is a software delay function for which the delay time can be controlled by the argument passed to the function. *delay10(unsigned int t)* is a shorter version of *delay* function. These software delay functions are implemented with blank iterations. *optfft(signed int real1[128], signed int real2[128], signed int image[128], signed image2[128])* is the FFT function. It is modified from the FFT function provided on the class website. Specifically, the original version of the FFT fuction accepts two signed int arrays each consists of 256 integers. Then the content of this function is also modified based on the modification on the prototype. This modification is because of the limitation of the linker file, as shown in the figure below.



**Figure 8,** DATABANK part in the linker file for PIC18F452

As we can see from the figure, the maximum size of global int array the PIC, or the compiler supports is 256 bytes, or 128 intergers. Also note that, to use DATABANK, a line

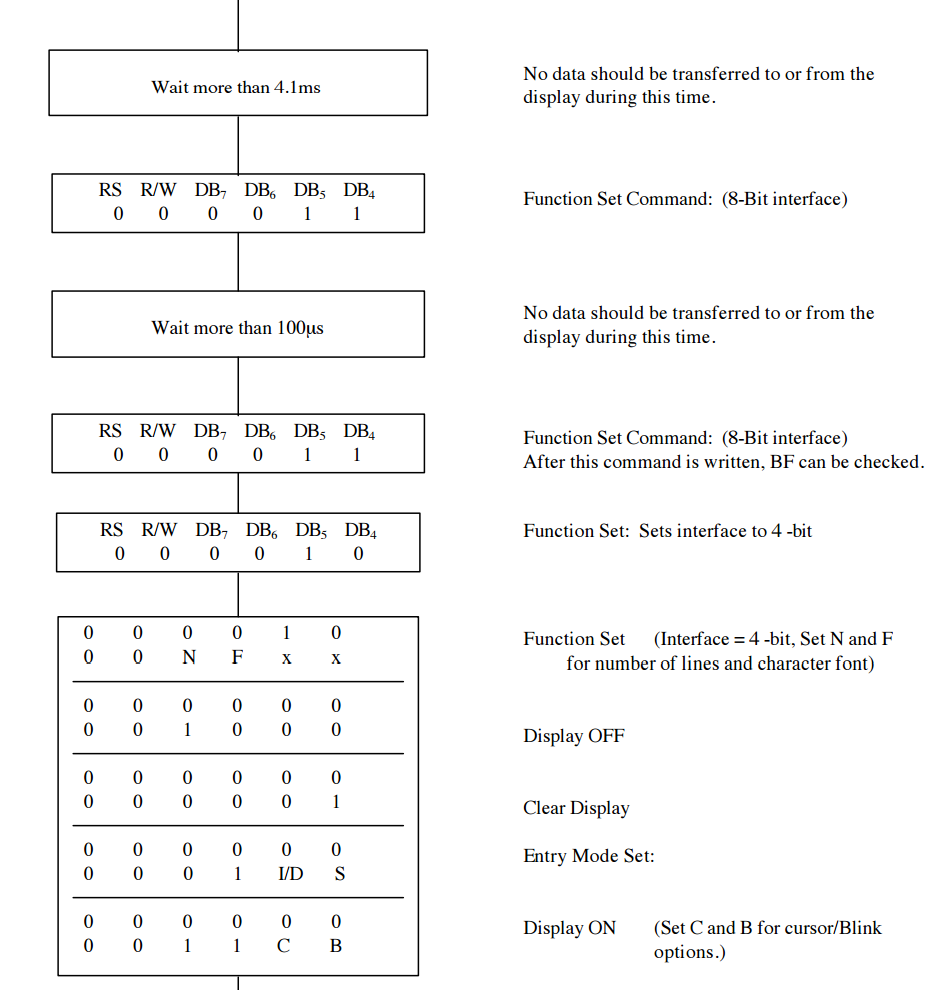
*#pragma udata gprX*

should be used immediately before the defination of the array. Although as shown in the user manual, larger arrays can be created by modifying the linker file, this solution is not used in our system since this will decrease the speed of the chip. Also note that, to store the FFT table in the chip, lines

*#pragma romdata bigvector*

*rom const T xx[N]={...}*

are used, where *T* is the type of elements in the array, *xx* is the name of the array, and *N* is the number of elements in the array. These two lines make the compiler and programmer to store the array as constant in ROM on the chip. Functions with names contain “LCD” are all LCD drivers. Their names are straightforward: *initLCD(void)* to initialize the LCD, *LCDputchar(char c)* to print a character on the LCD, *LCDprint(char \*s)* to print a string or char array on the LCD, *LCDconsprint(const rom char \*s)* to print a constant string or constant char array on the LCD, *LCDiprint(int i)* to print an integer on the LCD, *LCD1st* to move the cursor to the start of the first line, and *LCD2nd* to move the cursor the the start of the second line. LCD functions are all 4-bit version, to save pins on PIC. All functions are implemented according to the datasheet. The initialization flow chart provided by the datasheet is shown below:



**Figure 9,** 4-bit version LCD initialization procedure in datasheet

Also according to the datasheet, to move the cursor to the beginning position of first line on the LCD, RAM address should point to the beginning of DD RAM, and signal 0(RS) 0(R/W) 1(D7)1 00 0000(D0) need to be send to the LCD. Similarly, to move the cursor to the beginning position of second line on the LCD, RAM address should point to the beginning of CG RAM, and signal 0(RS) 0(R/W) 0(D7)1 00 0000(D0) need to be send to the LCD. Since the LCD drivers implemented in this system are all 4-bit version, each 8-bits data or control signals should be send in two times. Specifically, the first four bits sent to the LCD is regarded as higher 4 bits and the second four bits are lower 4 bits data or control signals. The characters are printed to the LCD with the same signal as their ASCII code. But different from sending control signals, to transmit data to the LCD, RS signal should be set as constant High. The rest of the functions with names contain “print” are used to transmit different types of variables through UART to PC, or print to Hyperterm. All of these functions are implemented based on the function *putsUSART* in the built-in library <USART.h>. *iprint(int i)* prints an integer to Hyperterm, *iprintln(int i)* prints an integer with line feed signals to Hyperterm, *dprint(double d)* print an double to Hyperterm, *dprintln(double d)* prints an double with line feed characters to Hyperterm, and *xprintln(int i)* prints an interger in hex representation. *ave(void)* is used to calculate the average value of the most recent ten measurements. Note that, the parameter of *ave* function is void, because it uses global variables (a double array consists of 10 numbers) for calculation. We used the results of measurements as global variables for convenvience. Also note that, while *ave* function is not in requirement of this project, it is only implemented in the master instrument, because this function directly print the results of calculation to Hyperterm, but the slave cannot communicate directly with the PC. Functions *writeSRAM(char addr, char c)* and *readSRAM(char addr)* are used to write to and read from the SRAM. To save GPIO pins, the SRAM is connected with the system through two SIPO shift registers, one PISO shift register and one tristate buffer, as will be discussed in Hardware Implementation section. Thus the *writeSRAM* and the *readSRAM* functions mainly manipulates control signals for those shift registers, the tristate buffer, and the SRAM. Function *writeSRAM* first shift address to a SIPO shift register. After address have been changed, then data is shifted to another SIPO shift register. After all set up, the PIC will send out signals to enable the tristate buffer and write state of the SRAM. Function *readSRAM* operates in a very similar way - first change address then shift data. But there are still two main differences. First, the shift register used is PISO, since it loads data from parallel outpur of the SRAM and sends serial data to the PIC. And second, the control signal for SRAM continuously at read state in this reading process, since for reading procedure, there are no issues such as unintentional overwriting if the control signal is not carefully settled. *slaveselect(int s)*, as its name implies, selects slave, and is also only implemented on the master device. It outputs the serial number of a slave need to be selected with three bits of binary number. Thus in theory this system can support up to eight slaves, though optimizaiton may needed if more than seven slaves are connected to the master device. However, since this system is only required to support six slaves, the abscence of this optimization is acceptable.

Most of interrupts are set up using bitwise configuration, while some of them are set by passing parameters. For bitwise configuration, to set up an interrupt, in most cases the only setting needed is to assign 1 to *XX(N)bits.XXIE*. For some others such as UART, setting interrupt priority is also needed. For configurations by passing parameters, usually *XX\_XX\_INT\_ON* should be passed to *OpenXxx* functions. For example, to setup timer0 with interrupt on can be written as (the real code is much longer, with other configurations):

*OpenTimer0 (TIMER\_INT\_ON)*

If any interrupt bit, all with names *XX(N)bits.XXIF*, is set, based on the interrupt priority settings, the program counter will go to interrupt handlers. For PIC, there are two interrupt handler code blocks, the high priority interrupt handler code is at address 0x8, and the low priority interrupt hander code is at address 0x18. As an example, interrupt handlers in our system are defined as below:

*#pragma code high\_interrupt=0x08*

*void high\_int(){...}*

*#pragma code*

*#pragma code low\_interrupt=0x18*

*void low\_int(){...}*

*#pragma code*

In our system, low priority interrupt is not used. The high priority interrupt handler (high\_int in our system) has a very limited space, 16 bytes. A common solution to this problem is to use a line of nested assembly code and redirect the instruction pointer to a function, which has enough space in which interrupts can be distinguished and dealt with separately. Still, as an example, our solution is shown below:

Assembly Code:

*\_asm GOTO high\_isr \_endasm*

Destination Function of Redirection:

*#pragma interrupt high\_isr*

*void high\_isr (void){*

*if(INTCONbits.TMR0IF){...}*

*if(PIR1bits.RCIF){...}*

*if(INTCON1bits.INT0F){...}*

*}*

In curly brackets are real handler function calls. Note that, since interrupt handler code is only need to be executed once, after dealing with the tasks, either the high priority handler function (*high\_isr* function in our system) or its sub functions should clear the interruption bit. *rx\_handler(void)* is used as the handler upon UART reception (RX). Note that, although the UART reception interrupt is removed from the slave instrument, which makes sense, since the slave instruments cannot communicate with the PC, UART it self is not removed from the slave device code for convinience. In our system, UART reception interrupt handler deals with the commands come from the PC. In detail, after recieving certain signals (each signal consists of two characters), it will set variable *Measure* to certain value. Note that, the variable *Measure* does NOT only control the current measurement, but also other states of this system, including the master and the slaves, according to the command from the PC. In another word, it acts more like a “general flag”. For example, if *Measure* is set to 0, the system will print current event statistics to Hyperterm; if *Measure* is set to 1, then the master will begin to measure low frequency. For the nigh states (measurements, event output, and a slave select state) of the master, variable *Measure* has values 0 to 8. For the control state for the slaves, ASCII code of letters are used for convinience. Note that, there will not be conflict between the two, since range 0 to 7 does not overlap with ASCII code of any English letter. A full state code table of USART (master only) is shown below:

|  |  |  |
| --- | --- | --- |
| Measure (State Code) | CMD | Meaning |
| 0 | SU | SUMMARY |
| 1 | FL | LOW FREQ |
| 2 | FH | HIGH FREQ |
| 3 | PS | SHORT PERIOD |
| 4 | PL | LONG PERIOD |
| 5 | IS | SHORT INTERVAL |
| 6 | IL | LONG INTERVAL |
| 7 | SP | SPECTRUM |
| 8 | S# | SELECT SLAVE # |
| ‘A’ ~ ‘J’ | RA ~ RJ | TRANSMIT ‘A’ ~ ‘J’  TO THE SLAVE |

**Table 1,** State Code of the Master Instrument

The handler for timer0, *timer0\_handler(void)*, works in the same way. Every time the timer overflow, it will trigger an interrupt, that is, set *INTCONbits.TMR0IF* to 1. Timer0 times the time interval for the frequency calculation of input square signal. Every time timer0 overflows, which is a relatively stable time interval, if the current system measurement state is frequency or period, then the PIC will read the value of the counter from the shift register. *int0handler(void)* is a handler for a interrupt triggered by the positive edge of input pin RB0/INT0. This interrupt is used to time the interval of the nonperiodic signal. *spi\_handler(void)* is a handler in code on slave instruments for receiving message. When the master is sending to the slave any message, this interrupt will be triggered and the slave will receive the message. Same as the master, slaves also have state code, and a state code table is shown below:

|  |  |  |
| --- | --- | --- |
| Measure (State Code) | MSG From Master | Meaning |
| 1 | A | LOW FREQ |
| 2 | B | HIGH FREQ |
| 3 | C | SHORT PERIOD |
| 4 | D | LONG PERIOD |
| 5 | E | SHORT INTERVAL |
| 6 | F | LONG INTERVAL |
| 7 | G | SPECTRUM |
| 8 | H | RESERVED |
| 9 | I | SND MES RES TO MSTR |
| 10 | J | SND EVENTS TO MSTR |

**Table 2,** State Code of Each Slave Instruments

There are several small details in this system need to be mentioned here. Firstly, the ADC is on by default. So to use port A as digital inputs, certain amount of ADCs should be turned off, by setting the value of register *ADCON1bits.PCFG[3:0]*. In our case, we only need 1 ADC channel and thus *ADCON1bits.PCFG[3:0]* is set to 0b1110, which means only open ADC channel 0 and use VDD and VSS as reference. This way, we can use the maximun number of GPIO pins. Secondly, RA4 is an open-drain pin. Thus to get correct output logic level, a pull up resistor should be used. Finally, the code in interrupt handlers should not be very long. If it is very long, the other interrupts may be influence, and the system may crash. All notes in this paragraph are fully discussed in the Error Analysis section, please refer to Error Analysis part for more detailed information.

Finally, algorithms used in this system are mostly straightforward. For frequency measurement, the most important part is the hardware timer. Every time the hardware timer triggers and interrupt, which means a relatively constant time, T, has passed, the PIC reads the value of the counter, C, from a PISO shift register. Then the freqency is calculated using equation

*f = C/T*

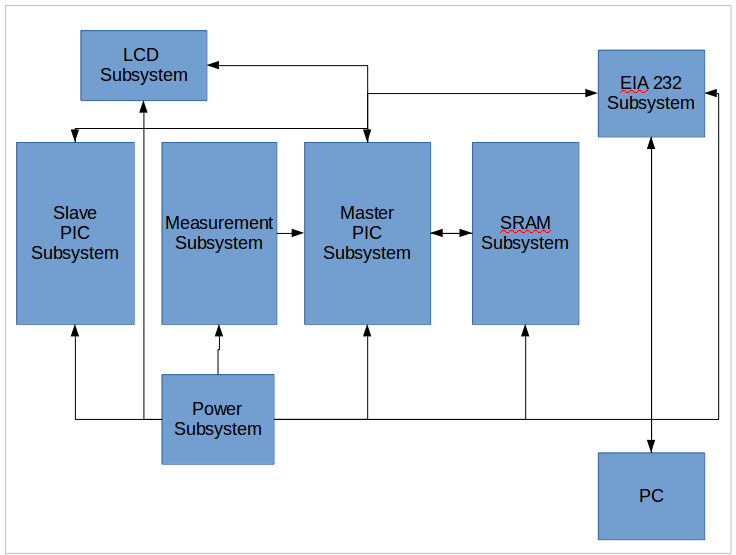
High freqency and low freqency are both calculated in this way. The only difference between the two is different bits of the counter (and thus different shift register) and different time interval are used. Specifically, as will be discussed in Hardware Implementation section, two *CD4040B* counters are connected in series, and low frequency measurement reads values from bit 8 to bit 1 in around 2.5 seconds time interval, while high frequency measurement reads values from bit 20 to bit 13 in approximately 0.83 seconds. In theory, the range of low frequency measurement is 0-100 Hz, with resolution of around 0.1 Hz. And the system can measure high frequency signals up to 1.26 MHz. Short period is calculated from frequency by equation

*P=1/f*

Since both short period measurements stated in the requirement are too long for the high freqency counter, the low frequency counter is used to calculate the period. The long period measurement is done using interrupt and timer, in the same way as the interval measurements discussed later. We first used calculation as short period to measure long period, however we finally chose to measure long period in this way, since the result was inaccurate, as discussed in the Error Analysis section. The range of long period is up to 1 second, with resolution of 0.01 second. and the small period measurement subsystem can measure period up to 10 ms, with resolution of 0.01 ms. For non periodic signals, interrupt is applied to measure their intervals. *INT0* will be triggered upon the rising edge of the input signal. Then the interrupt handler will reset the timer and go into a while loop waiting for the ending of the signal. After the signal ends, the system will read the value of the timer, and this value, after scale, will be the length of the singal. To measure long and short time interval, two timers, timer0 and timer3, with same clock source, internal instruction clock, but different prescale value are used. The measurement ranges should be, up to around 2 second for long time interval with resolution of 0.01 s, and up to greater than 10 ms with 0.01 ms resolution for short time interval. The kernel of spectrum measurement is ADC and FFT. FFT accepts 256 ADC output samples shifted and scaled to [-31, 32], and a empty array of 256 integers. Since output ADC and the actual voltage value has a linear relationship, a very simple shift and scale can be conducted. To get a even and very short sample interval, software delay, rather than hardware delay, is used. However, although the average sampling rate is always about 4736 samples per second, the system cannot maintain a constant high sampling rate, as will be fully discussed in Error Analysis section. The spectrum measurement system can reach a quitely good accuracy in frequency range 500-1000 Hz, as stated in the requirements. Lastly, an unsigned integer array is used to store the number of events. For each measurement, the number of events can count up to 65535, which also satisfies the reqirements.

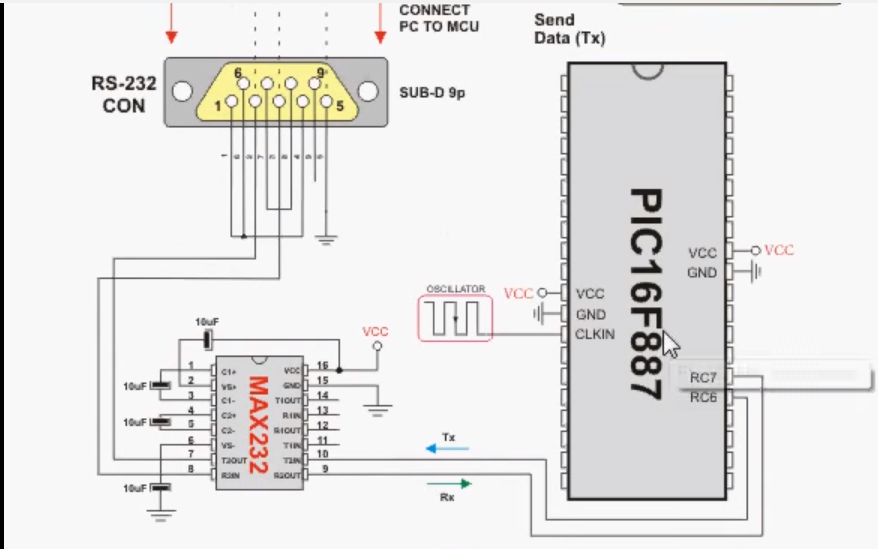
**Hardware Implementation**

The hardware of our design of the microprocessor based testing instrument/system mainly contains 9 parts, below is the figure of the final detailed block diagram of our systemm, figures of detailed block diagrams of subsystems, and the detailed interconnections between each components/modules of our design :



**Figure 10,** Final detailed diagram of the whole system

**Connection between the LCD & EIA232 & Local PC subsystems**



**Figure 11,** Communication subsystem

**Detailed connection of the Measurement subsystem**



**Figure 12,** Measurement subsystem(frequency)

**Detailed connection of the Master and the Slave**



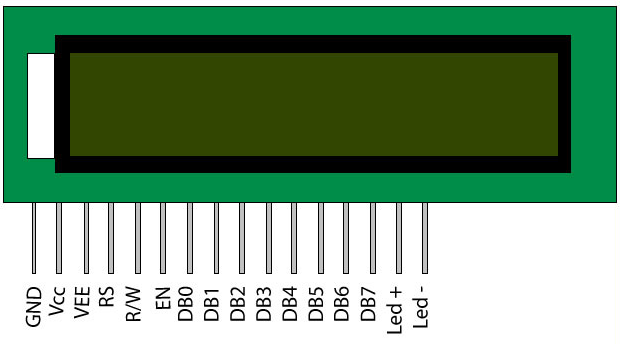
**Figure 13,** SPI network subsystem

**Detailed connection of the SRAM subsystem**



**Figure 14,** SRAM subsystem

**LCD**



**Figure 15,** Datasheet of the LCD

Here is the detailed interconnections of the LCD of our system:

GND -- Conntect to the ground

Vcc -- Connect to the power supply, +5V

Vee -- Connect to the ground in series with a 10k potentiometer

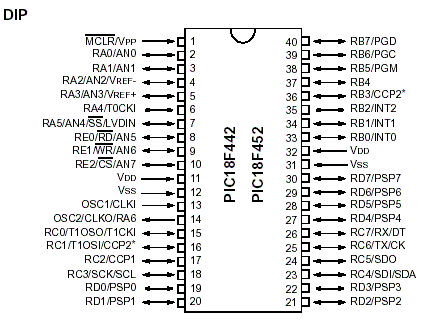
RS -- Connect to the gpio on the PIC(master)

R/W -- Connect to the gpio on the PIC(master)

En -- Connect to the gpio on the PIC(master)

DB0-DB7 -- Connect to the gpio on the PIC(master)

**PIC(master) (see the next page)**



**Figure 16,** Datasheet of the PIC18F452

Here is the detailed interconnections of the PIC(master) of our system:

RB2(gpio) -- Connect to the Enable of the LCD

RB3(gpio) -- Connect to the R/W of the LCD

RB4(gpio) -- Connect to the RS of the LCD

RB1(gpio) -- Connect to the DS, which is the data pin of the Shift Register 1 between counter and the PIC

RB6(gpio) -- Connect to the STCP, which is the latch pin of the Shift Register 1 used for the time of shift bit of the buffer in the Shift Register 1 between the counter and the PIC

RB7(gpio) -- Connect to the SHCP, which is the clock pin of the Shift Register 1 between the counter and the PIC, used for the time of parallel out all 8-bits data to the LCD

RD6 -- Connect to the Shift/Load pin of the Shift Register 1 between the counter and the PIC

RD7 -- Connect to the clk inhibit pin of the Shift Register 1 between the counter and the PIC

RD5 -- Connect to the rst/reset pin of the Shift Register 1 between the counter and the PIC

RD3 -- Connect to the Qh pin, which is serial output pin of the Shift Register 1 between the SRAM and the PIC

RA1 -- Connect to the serial input and gate pin, with two inputs A, B in serial, of the Shift Register 2 between the SRAM the PIC.

RE0 -- Connect to the clk pin of the Shift Register 2 between the SRAM and the PIC.

RE1 -- Connect to the clk inhibit pin of the Shift Register 1 between the SRAM and the PIC.

RE2 -- Connect to the SH/nLO pin of the Shift Register 1 between the SRAM and the PIC.

RD4 -- Connect to the eight enbles pins of the Tri-state between the Shift Register 2 and the SRAM.

RA6 -- Connect to the clock pin of the Shift Register 1 beteen the counter and the PIC.

RC0 -- Connect to the serial input data pin of the Shift Register 2 between the SRAM and the PIC.

RC1 -- Connect to the nWE pin of the SRAM

RC2 -- Connect to the nOE pin of the SRAM

OSC1 -- Connect to the clock pin of the Crystal Oscillator

Tx -- Connect to the Rx pin of the MAX232

Rx -- Connect to the Tx pin of the MAx232

Vdd -- Connect to the power supply and the Vdd pin of the PICKIT 3, +5V

Vss -- Connect to the ground and the Vss pin of the PICKIT 3.

nMCLR/Vpp -- Connect to the Vpp pin of the PICKIT 3

PGD -- Connect to the PGD of the PICKIT 3

PGC -- Connect to the PGC of the PICKIT 3

RA4 -- Connect to the A0, which is the input of the 3-8 decoder

RA5 -- Connect to the A1, which is the input of the 3-8 decoder

RB1 -- Connect to the A2, which is the input of the 3-8 decoder

RD0 -- Connect to the output of the Shift Register 1

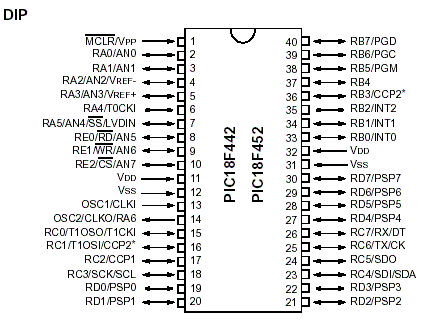
RD1 -- Connect to the output of the SHift Register 1

SDI -- Connect to the SDO of the PIC(slave)

SDO -- Connect to the SDI of the PIC(slave)

SCK -- Connect to the SCK of the PIC(slave)

**PIC(slave)**



**Figure 17,** Datasheet of the PIC18F452

Here is the detailed interconnections of the PIC(slave) of our system:

nSS(ss low true) -- Connect to the output of the 3-8 decoder

SDI -- Connect to the SDO of the PIC(master)

SDO -- Connect to the SDI of the PIC(master)

SCK -- COnnect to the SCK of the PIC(master)

Vdd -- Connect to the power supply and the Vdd pin of the PICKIT 3, +5V

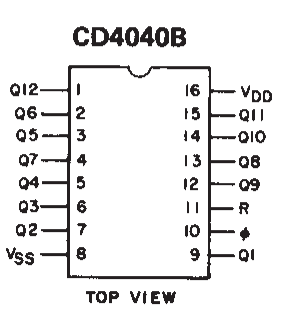
Vss -- Connect to the ground and the Vss pin of the PICKIT 3.

nMCLR/Vpp -- Connect to the Vpp pin of the PICKIT 3

PGD -- Connect to the PGD of the PICKIT 3

PGC -- Connect to the PGC of the PICKIT 3

**Counter**



**Figure 18,** Datasheet of the CD4040B

Here is the detailed interconnections of the Counter of our system:

Q1-Q8 of the first counter-- Connect to the 8 inputs of the Shift Register 1

Q7-Q12 of the first counter + Q1-Q2 of the second counter -- Connect to the 8 inputs of the Shift Register 1

Vss -- Connect to the ground

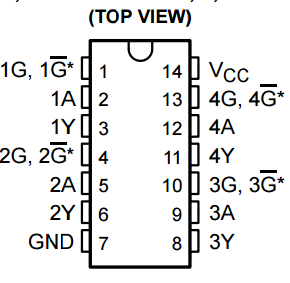
Vdd -- Connect to the Vcc, +5V

R -- Connect to the ground

Clk(pin 10 in the datasheet) -- Connect to the system input, which is connected to the function generator

The Q12 of the first counter is connected to the Clk pin of the second counter.

**Tri-state**



**Figure 19,** Datasheet of the 74LS128

We used two tri-states for 8-bit of data transmitted to the SRAM, with each tri-state for 4-bit

Here is the detailed interconnections of the tri-state of our system:

Each input of the Tri-state is connected to the output of the Shift Register 2(SIPO)

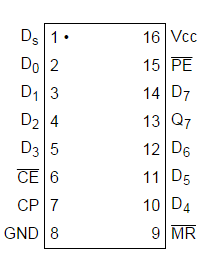
Each control signal of the Tri-state is connected to the RD4 of PIC(master), which is a gpio.

Each output of the Tri-state is connect to the input data pin of the SRAM.

GND -- Connect to the Vss

Vcc -- Connect to the power Vcc, +5V

**Shift Register 1(74166)**



**Figure 20,** Datasheet of the 74LS166

We used three 74LS166 for our system, two for connections between the counter and the PIC(master), and the other one for the connection between the SRAM and the PIC(master).

Here is the detailed interconnections of the 74LS166 of our system:

D0-D7 -- Connect to the 8 bit inputs of the counter & the 8 bit input of the data of the SRAM.

nMR(nCLR) -- Connect to the ground.

nCE(CLK INH) -- Connect to the RE1 of the PIC(master), which is a gpio.

CP(CLK) -- Connect to the RE0 of the PIC(master), which is a gpio.

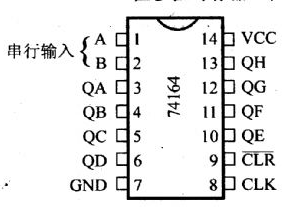
nPE(SH/nLD) -- Connect to the RE2 of the PIC(master), which is a gpio.

Ds -- Connect to the ground.

GND -- Connect to the ground.

Vcc -- Connect to the Vcc, +5V

**Shift Register 2(74164)**



**Figure 21,** Datasheet of the 74LS164

We used two 74LS164 for our system, one for the connection between the PIC(master) and the SRAM the other for the connection between the PIC(master) and the tri-state. Here is the detailed interconnections of the 74LS164 of our system:

A-B -- Connected together in series, one connect to the RA1(for connection between PIC & SRAM), one connect to the RC0(for connection between PIC & tri-state).

QA-QH -- Outputs of the 74LS164, one connect to the 8 bit address pin of the SRAM, the other connect to the 8 bit inputs of the tri-state.

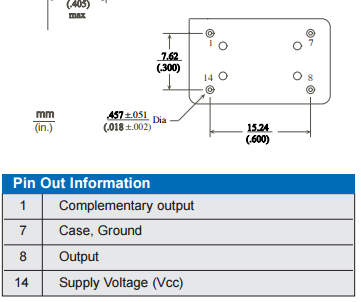
nCLR -- Connect to the ground

CLK -- Connect to the RE0 of the PIC(master), which is a gpio.

Vcc -- Connect to the Vcc, +5V

GND -- Connect to the ground.

**Crystal Oscillator**



**Figure 22,** Datasheet of the Crystal Oscillator

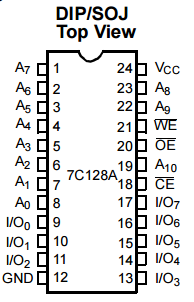
Here is the detailed interconnections of the Crystal Ocillator of our system:

Pin 1 -- Connect to the OSC1/CLK1 of the PIC(master)

Pin 7 -- Connect to the ground

Pin 14 -- Connect to the power supply, +5V

**SRAM**



**Figure 23,** Datasheet of the SRAM

Here is the detailed interconnections of the SRAM of our system:

A0-A7 -- Address of the SRAM, each connected to the 8 bit outputs of the 74LS164

I/O0-I/O7 -- Data of the SRAM, each connected to the outputs of the tri-states(labeled Y in the datasheet above)

nWe -- Connect to the RC1 of the PIC(master)

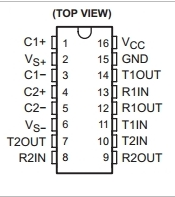
nOE -- Connect to the RC2 of the PIC(master)

nCS -- Connect to the ground

Vcc -- Connect to the power supply, +5V

GND -- Connect to the ground

**MAX232**



**Figure 24,** Datasheet of the MAX232

Here is the detailed interconnections of the MAX232 of our system:

C1+ connect to C1- in series with a 1uF capacitor.

Vs+ connect to Vcc in series with a 1uF capacitor.

C2+ connect to C2- in series with a 1uF capacitor.

Vs- connect to the ground in series with a 1uF capacitor.

T2OUT - connect to the male side connected to the port 2 of RS232

R2IN -- connect to the male side connected to the port 3 of RS232

Vcc -- connect to the power supply, +5V

GND -- connect to the ground

T2IN -- connect to the RC6, RX of the PIC(master)

R2OUT -- connect to the RC7, TX of the PIC(master)

**EIA232**

Here is the detailed interconnections of the EIA232 of our system:

Port 1,4 and 6 in series.

Port 2 connect to the T2OUT of the MAX232

Port 3 connect to the R2IN

Port 5 connect to the ground

Port 7 and 8 in series.

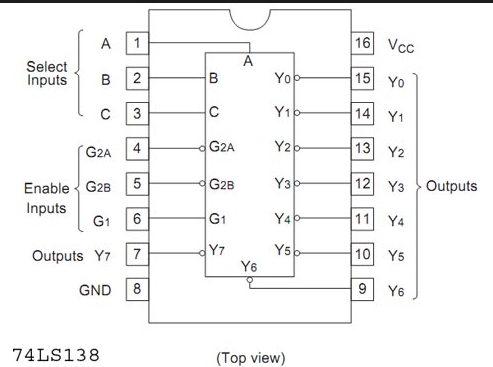
Port 9 unused.

Below is the correct connection of our EIA232 cable between the port side connecting to the PC and the male side connected to the MAX232:

|  |  |
| --- | --- |
| **Male Side** | **Port Number** |
| **Red** | **2** |
| **Blue** | **7** |
| **Brown** | **1** |
| **Orange** | **3** |
| **Yellow** | **4** |
| **Green** | **5** |
| **Silver** | **GND** |
| **Black** | **8** |

**Table 3,** Datasheet of the EIA232 cable used

**3-8 Decoder**



**Figure 25,** Datasheet of the 74LS138

Here is the detailed interconnections of the PIC(master) of our system:

A -- Connect to the RA4 of the PIC(master), one input of the decoder

B -- Connect to the RA5 of the PIC(master), one input of the decoder

C -- Connect to the RB1 of the PIC(master), one input of the decoder

Y1 -- Connect to the nSS(low true SS pin) of the PIC(slave)

Y0 & Y2 to Y7 -- unused

G2A -- Connect to the ground, one enable input of the decoder

G2B -- Connect to the ground, one enable input of the decoder

G1 -- Connect to the Vcc, +5V

GND -- Connect to the ground

Vcc -- Connect to the Vcc, +5V

**Testing**

**Test Plan**

Since this lab is very much complex and contains many hardwares so the testing of the whole system must be divided into serveral parts, and they should be tested one by one.

The parts are:

Master PIC

LCD subsystem

EIA 232 connection to PC

Couter subsystem (Measurement subsystem)

SRAM subsystem

Slave PIC

**For master PIC:** the main purpose is to make sure this PIC correctly accepts measured data from counter subsystem and print correct infomation onto the LCD module.

**For LCD subsystem:** testing for this part will focus on making sure the LCD successfully gets initialized and any input from master PIC should correctly appear on the LCD.

**For EIA 232 subsystem:** EIA 232 connction is critical for data exchange between PC and this lab system. The testing for this part is to make sure that data is correctly sent/ received between lab system and PC via HyperTerminal software.

**For counter subsystem:** this subsystem counts how many ups and downs in a given signal during a certain time period (defined by master PIC). The main purpose of testing this subsystem is to make sure that in, for example, one second, the counter should return a prallel output of 300 if the signal to be measured does have a frequency of 300Hz.

**For SRAM subsystem:** the testing of this part is very much similar to the process of last lab. The test for this subsystem should make sure that the SRAM is capable of storing in and reading out data.

**For slave PIC subsystem:** the slave PIC should communicate with master PIC via SPI interface. Although in this lab only one slave is installed, the system should be able to choose which one of the, at most, 6 slaves, to communicate with the master PIC.

**Test Specification**

Since the master PIC outputs two major types of data: serial/parallel actual data, or a single bit control signal such as slave select, it is reasonable to test them in different techniques.

**LCD Subsystem**

For testing this subsystem, the easiest and most efficient way is sending different strings to the LCD and let it display them out. Also the LCD should be capable of changing line of printing and clear the screen upon receiving certain instructions.

**For EIA232 Subsystem**

This subsystem mainly serves as a communication bridge between PIC and PC, so a testing strategy is that, by programming instructions of sending string data to PC onto PIC, and instructions of receiving string data from PC, the PIC should be able to communicate with the PC with any string data. To make sure that this subsystem is stable, this test should be done at least 3 times in 3 cases specified below in Test Cases.

**For SRAM Subsystem**

The test of SRAM subsystem follow the same strategy of previous lab, that is, by writing 255 integers to the SRAM first (0-255 to address 0-255), the SRAM subsystem should be able to output these 255 integer later. Also, to ensure that the SRAM subsystem is stable, at least 3 times of read/write process should be done, and data to be saved is specified in the following Test Cases part.

**The Counter Subsystem**

Controlled by the PIC’s built-in clock, this system should be able to count how many ups and downs of the given signal during a certain time period, so to test this subsystem, we use the master PIC to trigger on and off the measuring interval, and collect the data output of the counter. The frequency of the input signal should be (counter output / time interval), and if this value is close to the actual input signal frequency, it means that the counter subsystem should be correctly working. Again, to ensure the stability of the subsystem, at least 3 different signals should be tested.

**Master PIC**

Since master PIC contains huge amount of pins and the wire connection to it is extrememly complicatied, in this report, unlike the process (test each of the pin output) we did in the very first designing phase, we will be testing the master PIC by seeing that if the rest parts of the system connected to it can properly work.

**Slave PIC Subsystem**

Since the functionalities of slave PIC is very similar to the master PIC, the main focus of testing here is make sure that master and slave PIC should be able to communicate through SPI interface. This can be done by using slave select and let slave and master send certain strings of data to each other. This sending/receiving process should be done at least 3 times with different data.

**Test Cases**

**LCD Subsystem**

As specified in Test Specification, the follwing instructions should be done for testing.

Print “Hello”, “This is LCD Testing”, “One more TESTING” on first line

Print “Hello”, “This is LCD Testing”, “One more TESTING” on second line

before each time the LCD prints a new string, it should CLEAR itself first

**EIA 232 Subsystem**

As specified above, the PC and PIC should communicate with following strings.

“Hello from PIC”, “Hello from PC”, “PIC copy”, “PC copy”, “PIC sending again”, ”PC sending again”

The PC should first send “Hello from PC”, and upon receiving this by PIC, the PIC should send “PIC copy” to the PC as comfirmation. Then the PIC sends “Hello from PIC” to PC, and PC send “PC copy”, “PC sending again” after receiving. And PIC will start second round of receiving & sending: after receiving “PC sending again”, PIC should send “PIC copy” and “PIC sending again”. At last the PC should send “Test complete” to HyperTerminal to end the test.

**SRAM Subsystem**

The testing is done by writing the following data into the specified addresses, and then read them out to the HyperTerminal.

Data Address

0-255 0-255

256-512 256-512

512-768 512-768

**The Counter (Frequency Measurement) Subsystem**

The test is done by giving input signal of following frequencies.

1 MHz

750KHz

100Hz

75Hz

**Master PIC**

Since the test is done by observing if the rest of the system works well, there is no Test Case for this part. However, by changing different input measurement signal the master PIC should produce different output to other subsystems, it means by oberserving the behaviours of other systems, we can generate a conclusion that if the master PIC is correctly working on processing different input.

**Slave PIC**

The main focus of testing is sending and receiving data to/from the master PIC with the following strings.

“Hello from slave”, “Hello from master”, “Slave copy”, “Master copy”

The slave PIC will first send “Hello from slave” the master PIC, upon receiving, master PIC should send “Master copy” to tell the slave PIC that it has received the data, and then master PIC send “Hello from master” to the slave PIC, and at last, slave send “Slave copy” to the master PIC. The master PIC should print “Test complete” to end the test.

**Presentation, Discussion, and Analysis of Results**

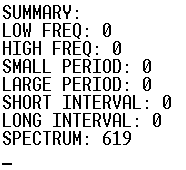
**LCD Subsystem**



**Figure 26.** LCD pattern of a empty signal input

Similar to the figure above, the LCD and successfully print strings specified in the Test Cases above and it can also clear its screen for displaying new data. After serveral times of printing, clearing, we conclude that this LCD subsystem is functional and stable.

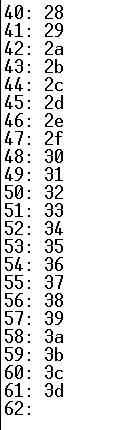
**EIA 232 Subsystem**



**Figure 27.** Summary report received by HyperTerminal

The EIA 232 Subsystem does ensures a communication between PIC and PC as shown in the figure above. Also through the testing described in Test Specification and Test Cases, the strings successfully got transmitted between PIC and PC, so we conclude that this subsystem is functional, and through serveral times of repeating the process, we believe that this subsystem is stable and free of bugs.

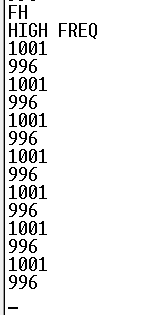
**SRAM Subsystem**



**Figure 28.** SRAM data output

As shown above in the figure (integer on left is address, and hex on the right is data stored), data is correctly stored and read to/from corresponding addresses. After repeating this read/write process with the 3 groups of data specified in Test Cases, we conclude that this SRAM subsystem is fully functional and stable.

**Counter Subsystem**



**Figure 29.** High frequency measurement of 1MHz

In the figure above, the unit of measuremnts is 1KHz.

As shown in the figure, the system gets very close measurements of the frequency of input signal (1MHz) by using the counter subsystem to count how many ups and downs of the signal during a small amount of time. Since the difference between measurements and actual frequency is lower than 5%, we conclude that this subsystem is functional, and reliable.

**Master PIC**

Since the testing of master PIC is based on the status of all other subsystems hooked onto it, and through the testing we found that the whole system works well and there is no significant error, so we conclude that the master PIC subsystem is functional. Also after many round of testing the whole system, we believe that the master PIC subsystem is stable and free of bugs.

**Slave PIC**

After trying sending string data specified in Test Cases above between master and slave PIC, we found that the communication between these two PIC is functional and stable. Also, by assigning different values in master PIC’s code, the master PIC can choose to activate a certian slave (although we only physically hooked one slave PIC, but the SS signal matches our expectation). Hence we conclude that, the slave PIC system can correctly work can produces no significant error during operation.

**Analysis of Errors**

1. UART configuration

After we connected circuit for UART, a sample code online was used. However, when we ran the code on PIC, its output to Hyperterm are wrong but not random. After checking the wiring carefully, we did not found any wrong connectiong. Then we went through the datasheet of PIC18F452, we found we need to change the second parameter of function *OpenUSART*, which is related to the baud rate. After calculating the baud rate using the equation given in the datasheet, as shown below

*FOSC/(16(x+1))=9600, where x is the second parameter*

in which, FOSC is the clock frequency, 20 M. The result is x=129. Thus we modified the second parameter to 129, and UART worked.

1. LCD initialization

At the beginning we planned to use a 8-bit serial in parallel out shifte register 74X595 to send data and command signals to the LCD, initializing the LCD in 8-bit mode. The reason a shift register was used is to save GPIO pins. However, after the circuit was built, the LCD subsystem did not work appropriately. Specifically, the cursor did show up but it moved around the LCD screen unintendedly. We checked the circuit and did not find wrong connections. We also tried to modify the code, which was originally written based on a worked LCD system in EE 472. Non of our effort resulted in any change in the behavior of the LCD. Finally, to save time, we decided to use LCD in 4-bit mode, and do not use the shift register anymore. This way, we need only one more GPIO pin, which is not many more, buf the system could be much easier to debug. So we changed the wiring and code. As we expected, the subsystem worked almost immediately without much debugging effort.

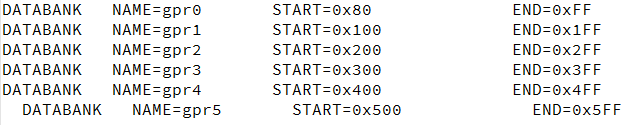
1. FFT data storage

As have been discussed in Software Implementation section, the FFT function needs a relatively large amount of data. These data includes two costant char array of 256 elements, one constant char array of 512 elements and two int array of 256 elements. We tried to store the constant arrays as function local variables, as global variables, and just as constant arrays as in the original version of the FFT function. However, all of the methods failed. Then we found a way to store large constant data online. That is, use lines

*#pragma romdata bigvector*

*rom const T xx[N]={...}*

where T is the type of elements in this data array, *xx* is the name of the array, and *N* is the number of elements in this array. So the issue with constant arrays are solved. However, the FFT function still requires two int arrays of 256 elements as its parameters, in which the elements are not constant. To solve this problem, we went though the user manual of C18 compiler. The solution the user manual provides for this issue is to modify the linker file. Specifically, there are lines in the linker file about DATABANK, as shown in the screen capture below:



**Figure 30,** Part of the Linker File

The user manual writes that, the programmer can write line

*#pragma udata gprX*

where *gprX* is the name of a DATABANK, immediately before the defination of the variable array to create a array of size up to 256 bytes. Also, by changing the start and/or end addresses of the DATABANK lines in the linker file, it is possible to create arrays even larger, at cost of slower processing speed. Since the integers occupies 16 bits on the PIC, a 256 elements integer array occupies 512 bytes storage. Thus we need to modify the linker file. However, this solution is not valid for us, since we are not administrator on computers in lab. Then we tried to use char array instead of integer array. But the result of the FFT function was wrong, though the input was tested to be correct. Later, under advice of the instructor, we modified the FFT function to accept fout integer arrays of 128 elements, which can be created legally with *gprX*, and this problem is finally resolved.

1. FFT sampling rate

In the process of solving the storage problem of FFT function, we once printed 256 ADC samples for FFT function several times to Hyperterm. We originally used a long floating point calculation immediately after sampling, as the way used in EE 472 lab. However, we found the sampling rate unsatisfying and unevn. We then modified the code and let the code did as less calculation and irrelavent operation (such as initalizing to imaginary number array passed to the FFT function) as possible, since the calculation speed does not matter comparing to the sample rate. Then the spectrum system worked satisfyingly.

1. SRAM reading

The SRAM reading results were wrong when tested at the beginning. Specifically, we tested the SRAM subsystem by writing 256 numbers in then reading them out and printing to Hyperterm, but the real reading result always “delayed” for one reading time. For example, while the correct reading value should be “0, 1, 2, 3, 4, 5, 6, 7, 8, 9...”, the real vaue were “X, 1, 2, 3, 4, 5, 6, 7, 8...”, where X was usually random. In this system, we used the same SRAM writing and reading procedure as we did in Lab 1 and the modified version of the shift register control code of frequency measurement subsystem (which is used to read counter values). After testing this subsystem with different test code and analyzing the results, we found that the writing procedure was correct since the numbers read out was different with different numbers written in, and if the numbers were only written in once and read out continuously, i.e. read from the beginning after one reading procedure, the first result in the reading procedure, i.e. the number “X” in the previous example, will be the largest number in the writing procedure, 0xFF in our case. Then we went through the code for reading procedure carefully and finally found the error was the wrong position of the line controlling the load signal to the shift register. So when we were reading in the testing process, the value loaded into the register was before the change of address. In another word, every time the value read out was the value in the previous address. After we moved the load signal control code to the code for changing address, the problem was resolved.

1. Summary could not be printed out

We first wrote our summary (event) code in UART RX interrupt handler code. Specifically, if “SU” command is typed into Hyperterm the UART RX interrupt would be triggered, and the handler should print out the statistics of the measurements so far. Our system only could print out the first two sentences in test. Based on the fact that all the functions called in the interrupt handler are tested and correct, we though this error may be caused by the long time delay in the interrupt handler, since staying long in the interrupt handler may interfere with other interrupts and leads to unintended behavior. Thus we modified the interrupt handler to only set a flag in the function itself, and the body part of the original handler was moved to the main function. Then the problem disappeared.

1. Inaccurate long period measurement in demo

In our demo, the long period measurement was not accurate. In detail, it can output 20 (unit here is 0.01 second) if the frequency is 5, which is correct, but the result becomes inaccurate with smaller frequency input, as in the demo, 2. We analyzed our algorithm, and thought the algorithm we used to measure long period was inappropriate. To calculate period, our calculation was based on equation

*P=1/f*

where *P* is the period and *f* is the frequency, since our frequency measurement subsystem was guaranteed to be working. We used a sampling rate of around 2.5 second per sample. This algorithm should work in thoery. However, after our analyzation, we realized this calculation may be inaccurate due to the precision of computation using PIC, especially with small numbers. Thus we modified the system and measure long period using interrupt and timer, in the same way as interval measurements. After this modification, the problem in the long period measurement subsystem was resolved.

1. Pin RB5 on PIC did not work

The pin RB5 did not work from the beginning. We firstly though it a hardware issue, and tried to avoid using it in our system. However, we finally ran out of GPIO pins and had to began to consider using RB5 pin. We tested another PIC, and found that the pin RB5 did not work as well on that new PIC with our test code. Then we realized that it may be an issue caused by default setting of the PIC. Therefore we went through the datasheet of the PIC and found that the LVP function of pin RB5 is on by default, and we had to turn it off in configuration. Thus we added a line

*#pragma config LVP=OFF*

to our code. Pin RB5 finally worked.

1. ADC influences port A input setting

When we were working on SPI, the ~SS (Slave Select, RA5) pin on slaves did not work properly. Specifically, while the ~SS pin on slaves should control the SPI function on slaves, i.e., communicate with master when ~SS pin is in Low state and not in High state, this pin actually did not have any control. For example, in tests, if ~SS pin was pulled down to GND, the slave can receive a message, then the slave still could receive that message if ~SS pin was pull up to VCC. We tested that pin with clock-like output and it worked. We then analyzed this error and, based on our experience with RB5 pin, as already discussed previously, we thought this maybe an error caused by interference of other functions of this pin. Then we went through the datasheet and found that the ADC channel can influence digital inputs. Therefore we setted the ADC channels (port A) so that only pin RA0 (ADC channel 0) acts as analog input, VDD and VSS as references for ADC, and all other pins as digital IO, by setting registers *ADCON1bits.PCFG[3:0]* to 0b1110. Then the ~SS pin worked properly.

1. Wrong pin RA4 Low

After we had configurated ADC, as discuessed above, we found pin RA4 consistently outputs Low. According to the datasheet, this pin should have been set to digital IO mode. We also tried another chip and found it output was also consistently Low. Then we searched on online and found this pin an open-drain pin. Thus we pulled it up to VCC through an 1K resistor and it worked.

Later when we were working on the SPI code, RA4 worked improperly again. Specifically, when the system was off, the RA4 pin was consistently High, which was proper since it was pulled up, but it was consistently Low when the system was on. Haven’t found any problem, we used pin RA2 instead. And then replaced all RA4 in the code with RA2. Finally the system worked.

1. SPI interrupt conflict with UART interrupt on slave instrument

After we added SPI receive interrupt to the slave system code, the systems stucks somewhere. Thus we inserted a simple “blinky” code (which makes a GPIO pin output High and Low voltage alternatively) in our code at different places for test purpose. Then we found code stops at line

*INTCONbits.GIEH=1;*

Which enables global interrupts. This means, the system went to ISR immediately after all interrupts were enabled, and a interrupt bit had then never been cleared. Since all the interrupts were handled and all the interrupt bits possibly being set are all cleared at the end of each handlers, we decided to put the global interrupts enable line to the very beginning of the code and started testing immediately after this line again. We later found that the code actually stucks after UART RX interrupt was enabled. We do not know the real reason of this problem, but our guess is that the SPI receive interrupt and UART RX interrupt may have some form of conflict. The solution was very simple, we just removed UART RZ interrupt from the slave system code, since UART is not needed in the slave system at all, it was included only for convinience.

1. SPI master reading / slave writing issue

When working on the master instrument reading data from the slave device, at start, only one character could be recieved. We checked the circuit and the code for the master instrument and the slave instruments, and did not find any error, given the fact that the SCK (Serial Clock) pin has signals continuously, and that after the first character was recieved, the system was then blocked at the next write operation. After we analyzed the communication system, we thought the protocol settings most suspicious. After tried several settings, we found that one fourth system clock, (1, 1) SPI mode, and sampling at the midle of the clock could guarantee correct data reception. And therefore this problem was resolved.

1. SPI communication weird behavior

After SPI system was entirely incorporated into the while system, after some simple controlled test, we found that only every time after VCC was shorted to the ground, the SPI system could work properly. Apparently some parts of the circuit forms a capacitor and/or a inductor large enough to affect the communication system. However, due to the scale of the project and the time left for the circuit, we could not apply controlled test device by device for the system. We did checked our hardware connection but did not find any error, which is reasonable since the circuit is able to produce correct result. Based on rough analysis, we thought the most possible influence comes from the capacitor and inductor the complex layer of wires. But later we reconsidered this problem, and found all what we used was the built-in SPI library, <spi.h>, which may not be completely safe, given the abscence of any clear documentation. In another word, we might used the functions in this library in an inappropriate way. Thus we rewrote the code for master sending / slave transmission on slave side, and this communication problem was resolved.

**Failure Mode Analysis**

1. Master-Slave communication error or not responding

May due to wrong slave selection or slave connection. Slave number should range from 0 to 8, and only connected slave should be selected for correct behavior of the whole system.

1. Output out larger than measurement range

This error is either caused by the wrong connection of input signal, or wrong selection of measurement mode.

1. Output smaller than measurement range

This error is either caused by the wrong connection of input signal, or wrong selection of measurement mode.

1. No output

May due to wrong connectino of input signal, wrong selection of measurement mode, and illegal or unsafe voltage or current value by power supply.

1. System stuck after running for a long time

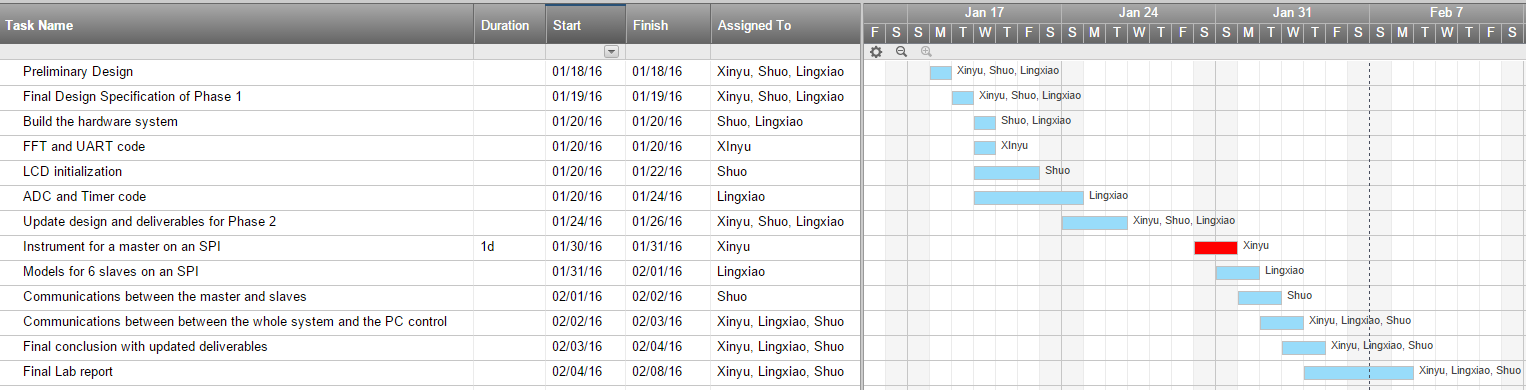
May due to illegal or unsafe voltage or current value by power supply, or Master-Slave communication error, as indicated in Faliure Mode part 1.

**Final Factory Cost(BOM)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Item | Description | Part Number | Quantity | Cost($) |
| 1 | Microprocessor | PIC18F452 | 1 | 8.00 |
| 2 | SRAM | CY7C128A | 1 | 4.00 |
| 3 | Tri-state | 74LS125A | 2 | 0.70 |
| 4 | LCD |  | 1 | 5.00 |
| 5 | MAX232 | MAX232N | 1 | 0.70 |
| 6 | Shift Register1 | 74LS164 | 2 | 0.60 |
| 7 | Shift Register2 | 74LS166 | 3 | 0.60 |
| 8 | Counter | CD4040B | 2 | 0.60 |
| 9 | Crystal Oscillator |  | 1 | 2.40 |
|  |  |  | Total | 22.6 |

**Table 4,** Final Factory Cost

**Final Updated Schedule(See Next Page)**



**Figure 31,** Final Updated Schedule

**Summary & Conclusion**

In conclusion, this project contains 2 phases. In the phase 1, we were introduced with two new tools, which are the PIC and the MPLAB. We understood and learnt how to use the PIC and built a stand-alone version of the microprocessor based testing instrument, including measurements of frequency, period, spectrum and time interval. The system we built in the phase 1 mainly contains four subsystems, which are communications subsystem, measurements subsystem and memory subsystem and display subsystem. We built the communication subsystem to make sure the communications betweent the PIC and the local PC work. We built the measurements subsystem to make sure the system is able to operate severval measurements specified. We built the memory subsystem to record the store data of recent measurements.We built the display subsystem used for displaying the results of each measurement. Then in the phase 2, we built an SPI interface and driver that will support a simple local area network with up to six other devices based on the system we built in the phase 1. During the design, we mainly used the hypeterminal and the logic analzyer to test our design. Finally, we implemented our design and it met all requirements in the specificiation with acceptable deviations/bias(less than 0.8%) from theoretical values. The experience we gained from using the PIC and the SPI network would be very useful for further projects.

**Appendices**

All figures & tables are uploaded into the dropbox.

**Contribution Table**

|  |  |
| --- | --- |
| **Content** | **Contributor** |
| Overall Design | Shuo Yin, Lingxiao Zhang, Xinyu Sui |
| **Phase 1** |  |
| UART and FFT | Xinyu Sui |
| LCD Module Manipulation | Shuo Yin |
| ADC and Timer Configuration | Lingxiao Zhang |
| **Phase 2** |  |
| SPI Interface | Xinyu Sui |
| Master and Slave Communication | Shuo Yin |
| Salve Configuration | Lingxiao Zhang |
| EIA 232 (PC Communication) | Shuo Yin, Lingxiao Zhang, Xinyu Sui |
| **Deliverables** | Shuo Yin, Lingxiao Zhang, Xinyu Sui |
| **Lab Report** | Shuo Yin, Lingxiao Zhang, Xinyu Sui |

**Signature**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**