

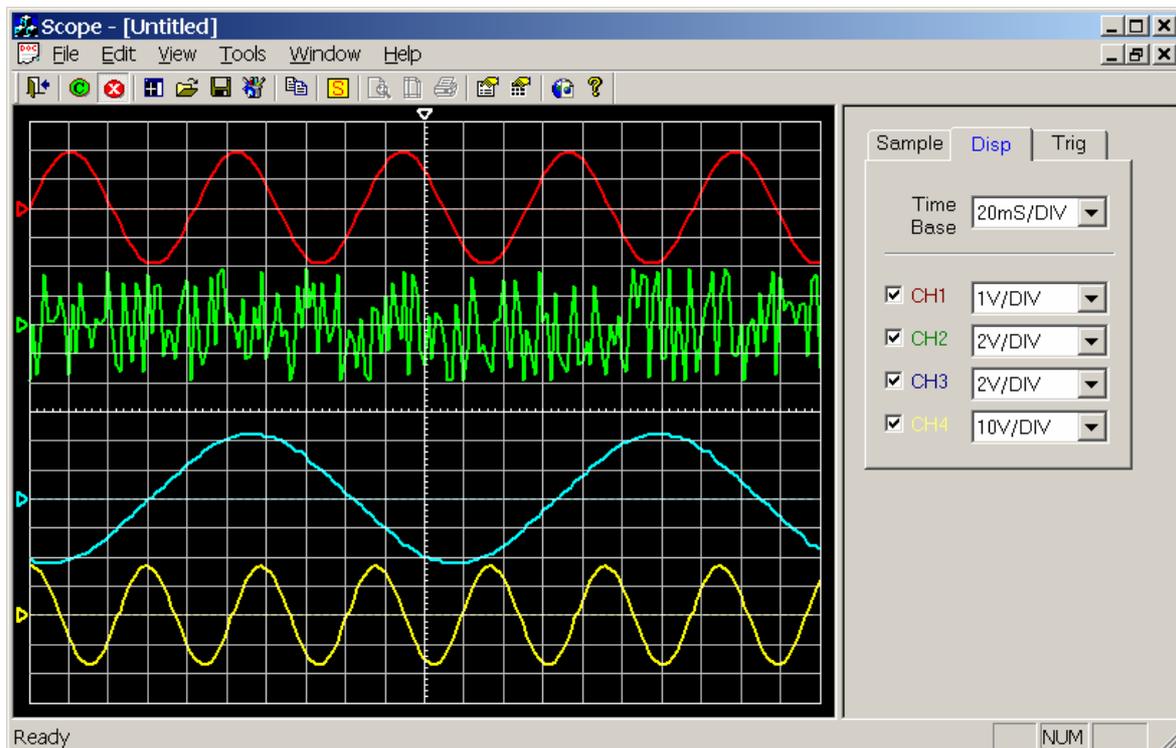
Final Year Project

Low cost PC-based quad channel real-time / storage oscilloscope

Final Report

Supervisors:

First: Dr. S. Katzen
Second: Dr C. Tuner



school of electrical and
mechanical engineering

COURSE: BEng (HONS) Electronic Systems
MODULE: EEE516J4 - Honours Project
BY: Colin K McCord
DATE: Friday, 21 June 2002

ACKNOWLEDGEMENTS

- Dr. S. Katzen:** For everything that he has done in helping the project along, and for offering support and encouraging an interest in the field of microprocessor / microcontroller engineering.
- Dr. C. Turner:** For everything that he has done in helping the project along, and for offering support and encouraging an interest in the field of object originated programming.
- Dr. Ning Li:** For the use of FG Wilson's electronic laboratory.

EXECUTIVE SUMMARY

The following report details the development and implementation of a low-cost PC based quad channel real-time / storage oscilloscope.

This project attempts to achieve the same functionality as a traditional oscilloscope, using a PIC microcontroller for data acquisition (including appropriate analogue circuitry) which transfers the data to the PC (via RS232). A Microsoft Windows based software application will then display the waveform as it would appear on a traditional CRT oscilloscope. This software application will have additional features not present on a traditional oscilloscope (e.g. printing / saving waveforms) with greater flexibility as additional features can be added as they developed without the need for new hardware.

The digital based oscilloscope should display very low frequency waveforms in real-time, but for higher frequency waveforms it is necessary to read a finite number of samples storing them into RAM. Once the memory is full (or the preset number of samples has been reached) the PIC will stop sampling and transfer the data to the PC, when ACK (acknowledgment) is received from the PC the PIC will start sampling again. This is known as a "Storage Oscilloscope", but there are disadvantages e.g. it's impossible to continuously monitor a waveform in real-time for more than the amount of samples that can be stored into the buffer as there would be gaps in the data.

Main Advantages of Digital over Analogue Scopes

- The ability to observe slow and very slow signals as a solid presentation on the screen.
- The ability to hold or retain a signal in memory for long periods.

Core Objectives

- Design and construct hardware required for data acquisition.
- Design of the real-time communication protocol.
- Design of a Windows application for displaying low-frequency waveforms in real-time.
- Design of a simulation program.
- Demonstrate the entire system working.

Further Development

- Add storage functionality increasing its sampling rate to at least 20 kHz.
- Using external ADC and external RAM, explain how it would be possible to monitor high-frequency waveforms?
- Investigate how far this storage technique can be pushed, what are the limits?

Summary

- Low-Cost PC Based Oscilloscope
- PIC16F877 microcontroller for data acquisition using 4 of the built-in ADCs (4 channel scope).
- Analogue circuitry to insure 0 to 5 V input to ADC (e.g. -100 to 100V converted to 0 to 5V, i.e. bipolar)
- Windows application (scope.exe) for displaying waveforms.
- Simulator program for testing graphical display, communication protocols and triggering techniques.

Possible Application

- Demonstration oscilloscope, using data projector and laptop.

Advantages

- Large high-resolution display.
- Windows / GUI advantage such as cut & paste into documents.
- Low-cost (less than £50)
- Software Upgradeable.

NOTATION

Abbreviations and symbols used in this report are listed alphabetically: -

%	Percentage
μ	Micro, 1×10^{-6}
∞	Infinity
ACK	Acknowledgement
ADC	Analogue to Digital Converter
ASM	Assembly level programming
BMP	Bitmap
bps	Bits per second, also known as the baud-rate
BRG	Baud Rate Generator
C	Capacitor
C	High-level programming language
C++	Object originated programming language
CCS	C compiler
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CRC	Cyclical Redundancy Check
CRT	Cathode Ray Tube
DDB	Device Dependent Bitmap
DIB	Device Independent Bitmap
DOS	Disk Operating System
F	Farads
$f_{\text{CUT_OFF}}$	High frequency cut-off
FERR	Framing Error
f_{STOP}	Start of filter stop band
GIF	Graphical Interchange Format
GND	Ground
GUI	Graphical User Interface, i.e. Microsoft Windows
Hz	hertz
I/O	Input / Output
IC	Integrated Circuit
JPEG	Joint Photographic Experts Group
k	Kilo, 1×10^3
LCD	Liquid crystal Display
m	Milli, 1×10^{-3}
M	Filter Order
M	Mega, 1×10^6
MB	Megabytes

MS	Microsoft
NAK	Negative Acknowledgment
OERR	Overflow Error
p	Pico, 1×10^{-9}
PC	IBM compatible personal computer
PIC	Peripheral Interface Controller
R	Resistor
RAM	Random Access Memory
RCSTA	Receive Status and Control Register
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RS232	Standard 9-pin PC serial port
RX	Receive
S	Switch
SCI	Serial Communications Interface
SPBRG	Serial Port Baud-Rate Generator
SPEN	Serial Port Enable bit
SPI	Serial Peripheral Interface
SREN	Single Receive Enable bit
TACQ	Amplifier Setting Time
T_C	Hold Capacitor Charging Time
T_{COFF}	Temperature Coefficient
TX	Transmit
TXSTA	Transmit Status and Control Register
UART	Universal Asynchronous Receiver Transmitter
USART	Universal Synchronous Asynchronous Receiver Transmitter
USB	Universal Serial Bus
V	Voltage
V_{CC}	Supply Voltage
V_{DD}	Supply Voltage
V_I	Input Voltage
V_{IN}	Input Voltage
V_o	Output Voltage
V_{OUT}	Output Voltage
V_{SS}	Ground
WWW	World Wide Web
X	Horizontal
XTAL	Crystal frequency
Y	Vertical
Ω	Ohms

TABLE OF CONTENTS

1.0. Project Background	Pages 1 to 3
2.0. Design Specifications	Page 4
2.1. Core Objectives	Page 4
2.2. Further Development	Page 4
3.0. Fundamentals	Pages 5 to 15
3.1. The Basic CRT Oscilloscope	Page 5
3.2. Digital Sampling Oscilloscopes	Page 6
3.3. Display Formats	Pages 6 to 8
3.4. Aliasing	Pages 8 to 12
3.4.1. Typical Low-Pass Filter Design Parameters	Page 9
3.4.2. Typical Low-Pass Filters	Pages 9 to 10
3.4.3. Software Generation of an Anti-Aliasing Filter	Pages 10 to 12
3.5. The PIC Microcontroller	Pages 12 to 13
3.6. RS232 Serial Interface	Pages 13 to 15
4.0. The PIC16F877 Microcontroller	Pages 16 to 20
4.1. Overview of the File Registers	Page 17
4.2. Overview of the 8-channel 10-bit ADC	Pages 18 to 19
4.3. Overview of the Hardware USART	Page 20
4.3.1. Baud-Rate Generator, BRG	Page 20
5.0. Hardware Development	Pages 21 to 29
5.1. Simplified Block Diagram	Pages 21 to 23
5.2. Digital Circuit Diagram	Pages 23 to 24
5.3. Analogue Circuit Diagram – Mark 1	Pages 24 to 26
5.4. Analogue Circuit Diagram – Mark 2	Pages 26 to 28
5.5. System Powering Circuit	Pages 28 to 29
5.6. Cost of Components	Page 29
6.0. Communication Protocol Development	Pages 30 to 42
6.1. Real-Time Mode	Pages 30 to 31
6.2. Storage Mode	Page 31
6.3. Control Protocol	Page 32
6.4. Frame Structure: Real-Time Mode	Pages 32 to 33
6.5. Sample Rate: Real-Time Mode	Pages 33 to 34
6.6. Frame Structure: Storage Mode	Pages 34 to 35
6.7. Sample Rate: Storage Mode	Page 35
6.8. CRC16 (Cyclical Redundancy Check) used in Storage Mode	Pages 36 to 41
6.8.1. Comparison of Raw Calculation CRC & Lookup Table CRC	Pages 38 to 41
6.9. Frame Structure: Control Protocol	Page 42
7.0. The Scope Program	Pages 43 to 85
7.1. Data Flow Diagram	Page 49
7.1.1. Data Flow Paths	Pages 49 to 50
7.2. The Grid	Pages 50 to 52
7.3. The Traces	Pages 52 to 58
7.4. Storing and Restoring the Previous Position of the Main Frame	Pages 58 to 59
7.5. Setting-Up RS232 Communications	Pages 60 to 62
7.6. Solving the Flickering Problem	Pages 62 to 63
7.7. Creating the Split Window View	Pages 63 to 64
7.8. Saving / Opening Scope Data	Pages 65 to 70
7.9. Copying the Scope Display to the Windows Clipboard	Pages 71 to 72
7.10. Exporting the Scope Display as a Bitmap (BMP) File	Pages 72 to 77

7.11. Owner Drawn Menus with Bitmaps	Pages 77 to 80
7.11.1. Integrating Brent Corkum's BCMenu Class with the Scope Program	Pages 78 to 80
7.12. Windows 95/98 Compatibility Problem (scope v1.008a, 20/12/2001)	Pages 81 to 83
7.13. Windows XP Compatibility Problem (scope v1.017a, 05/02/2002)	Pages 83 to 84
7.14. Additional Video Card Features That May Explain Low CPU Usage	Pages 84 to 85
8.0. The Simulator Program	Pages 86 to 89
8.1. How Channel Simulation is Achieved?	Pages 78 to 89
9.0. Test Program	Pages 90 to 95
9.1. User Manual	Pages 90 to 92
9.2. How is RS232 Communication Achieved?	Pages 93 to 95
10.0. PIC Software Development	Pages 96 to 113
10.1. Mark 1.c (Test RS232 Communications)	Pages 96 to 97
10.2. Mark 2.c (Fully Test RS232 Communications)	Page 97
10.3. Mark 3.c (Real-time Mode: 1 channel, fixed sampling delay)	Pages 97 to 98
10.4. Mark 4.c (Test RS232, Baud-Rate Set by dip-Switches)	Pages 99
10.5. Mark 5.c. (Fully Test RS232 Communications, Adjustable Baud-Rate)	Page 100
10.6. Mark6.c (Real-time mode: 1 channel, fixed delay, adjustable baud-rate)	Pages 100 to 101
10.7. Mark 7.c (Test Timer Interrupts)	Pages 102 to 103
10.8. Mark 8.c (Real-time mode: dual channel, fixed delay, adjustable baud-rate)	Pages 104 to 105
10.9. Mark 9.c (Real-time mode: four channels, chop, interrupt time-base)	Pages 105 to 109
10.10. Mark 10.c (Test External RAM Chip)	Pages 109 to 111
10.11. Mark 11.c (Main Program)	Pages 112 to 113
11.0 Test Results	Pages 114 to 147
11.1. Test Session 1: FG Wilson Lab (26/02/2002)	Pages 114 to 120
11.1.1. Test 1: Test RS232 Communications (TX only)	Pages 114 to 115
11.1.2. Test 2: Fully Test RS232 Communications	Pages 115
11.1.3. Test 3: Test ADC	Pages 115 to 119
11.1.4. Conclusion	Page 120
11.2. Test Session 2: FG Wilson Lab (05/03/2002)	Pages 120 to 131
11.2.1. Test 4: Test RS232 Communications (Baud Rate set by DIP Switches)	Page 120
11.2.2. Test 5: Fully Test RS232 Communications (Baud rate set by DIP Switches)	Page 121
11.2.3. Test 6: Test ADC, Scope Program and Real-time Communication Protocol	Pages 121 to 136
11.2.3.1. Sample Rate Changed to 2000Hz (mark6b.c)	Pages 125 to 129
11.2.3.2. Sample Rate Changed to 5000Hz (mark6c.c)	Pages 129 to 130
11.2.4. Test 7: Test Timer Interrupts	Page 130 to 131
11.2.5. Conclusion	Page 131
11.3. Test Session 3: FG Wilson Lab (22/03/2002)	Pages 131 to 147
11.3.1. Test 8: Dual Channel Test	Page 131
11.3.2. Test 9: Interrupt Drive Time-Base, Real-Time Sampling (Four Channels)	Pages 132 to 145
11.3.2.1. Dual Channel Operation Using Two Signal Generators	Pages 135 to 140
11.3.2.2. Quad Channel Operation	Pages 140 to 141
11.3.2.3. 5000Hz Sample Rate – Single Channel	Pages 141 to 145
11.3.3. Test 10: Test External RAM Chip	Pages 145 to 147
11.3.4. Test 11: 5V Regulator	Page 147
11.3.5. Test 12: Analogue Circuitry	Page 147
11.3.6. Conclusion	Page 147
12.0. Conclusions	Pages 148 to 151
13.0. Recommendations For Further Work	Pages 152 to 155
13.1. Control Protocol	Page 152
13.2. Storage Mode	Page 152
13.3. Multiple Serial Ports	Page 152
13.4. USB Transport Medium	Page 152
13.5. Bootstrap	Page 152 to 153

13.6. Virtual Channels	Page 153
13.7. Data Logging	Page 153
13.8. TCP / IP Internet Communications	Page 153
13.9. Direct Modem to Modem Communication	Pages 153 to 154
13.10. Printing of Scope Display	Page 154
13.11. An Alternative Method for Dealing with Aliasing	Pages 154 to 155
13.12. Taking Advantage of Aliasing	Page 155

14.0. Bibliography / References

Pages 156 to 158

A. CD ROM containing source code, datasheets, etc...

Page 159

NOTE: Full set of appendixes in separate booklet

TABLE OF FIGURES

1.0. Project Background	Page 1 to 3
1.0a. 20MHz Analogue Dual Trace (CRT), about £400 [W2]	Page 1
1.0b. 150MHz Analogue / Digital (CRT), about £1,600 [W1]	Page 1
1.0c. Diagram of a typical cathode-ray tube (CRT) construction	Page 1
1.0d. 100MHz 4 channel digital storage oscilloscope, about £1,800 [W2]	Page 2
1.0e. 100MHz handheld digital storage oscilloscope, about £1,000 [W2]	Page 2
1.0f. 100kHz dual channel PC based storage oscilloscope, about £250 [W2]	Page 2
1.0g. 20kHz single channel PC based storage oscilloscope, about £95 [W2]	Page 2
 3.0. Fundamentals	 Pages 5 to 15
3.1a. Block diagram of a basic CRT oscilloscope	Page 5
3.1b. Types of CRT oscilloscopes	Page 5
3.2a. Example showing how a sinewave is digitally sampled	Page 6
3.3a. Comparison of dot, pulse interpolator and sine interpolator displays	Page 7
3.4a. Demonstrating aliasing	Page 8
3.4.1a. The key low pass filter design parameters	Page 9
3.4.3a. Step 1: Open anti-aliasing wizard	Page 11
3.4.3b. Step 2: Enter 3dB cut-off frequency	Page 11
3.4.3c. Step 3: Enter sampling frequency	Page 11
3.4.3d. Step 4: Enter ADC resolution	Page 11
3.4.3e. Step 5: Ideal signal to noise ratio is calculated	Page 11
3.4.3f. Step 6: Click build Butterworth, the finish	Page 11
3.4.3g. Anti-aliasing filter designed by the anti-aliasing wizard	Page 12
3.5a. Simplified illustration of the von Neumann architecture	Page 12
3.5b. Simplified illustration of the Harvard architecture	Page 13
3.6a. Illustration of RS232, 1 driver and 1 receiver	Page 14
3.6b. 9-pin RS232 D-connector, pin signal description	Page 15
3.6c. Typical maximum distance modern line drivers/receivers can manage	Page 15
3.6d. Illustration of how data is transmitted over RS232	Page 14
 4.0. The PIC16F877 Microcontroller	 Pages 16 to 20
4.0a. Architecture of the PIC16F877 microcontroller [W9]	Page 16
4.1a. PIC16F877 register file map	Page 17
4.2a. Simplified block diagram of the PIC16F877 ADC module	Page 18
4.2b. Unknown eight placed on the scales	Page 19
4.2c. 8g weight placed on the pan, not too heavy (keep)	Page 19
4.2d. 4g weight placed on the pan, too heavy (remove)	Page 19
4.2e. 2g weight placed on the pan, not too heavy (keep)	Page 19
4.2f. 1g weight placed on the pan, too heavy (remove)	Page 19
4.2g. Unknown weight is about 10g (1010)	Page 19
 5.0. Hardware Development	 Pages 21 to 29
5.0a. Illustrates the operation of the analogue circuitry	Page 21
5.1a. Simplified block diagram	Page 22
5.2a. Digital circuit diagram (version 1.3, 19/03/2002)	Page 23
5.2b. Dip-switch configuration	Page 24
5.3a. Analogue circuit diagram – Mark 1 (14/12/2001)	Page 25
5.4a. Analogue circuit diagram – Mark 2 (19/03/2002)	Page 27
5.5a. System powering circuit	Page 28
 6.0. Communication Protocol Development	 Page 30 to 42
6.1a. Illustration of real-time mode with one channel	Page 30
6.1b. Illustration of real-time mode with four channels (chop mode)	Page 30
6.1c. Illustration of real-time mode with four channels (alternate mode)	Page 30
6.2a. Illustration of storage mode with one channel	Page 31
6.2b. Illustration of storage mode with four channels (chop mode)	Page 31
6.3a. Illustration of the control protocol	Page 32

6.4a. Frame structure: Real Time	Page 32
6.4b. Real-time frame structure example	Page 33
6.5a. Table showing the theoretical real-time sample rates (chop, 10-bit ADC)	Page 33
6.5b. Table showing the theoretical real-time sample rates (chop, 8-bit ADC)	Page 34
6.5c. Table showing the theoretical real-time sample rates (chop, 10-bit ADC, dual RS232)	Page 34
6.6a. Frame Structure: Storage Mode	Page 35
6.8.1a. Result of raw calculation CRC test	Page 39
6.8.1b. Result of lookup table CRC test	Page 41
6.9a. Frame structure: control protocol	Page 42
6.9b. SETUP_REG1	Page 42
6.9c. SETUP_REG2	Page 42

7.0. The Scope Program

Pages 43 to 85

7.0a. Screen dump of scope.exe (V1.041a, 19/04/2002)	Page 43
7.0b. Right click menu	Page 43
7.0c. Screen dump of scope.exe with side bar hidden	Page 44
7.0d. Screen dump of scope.exe demonstrating auto-resize	Page 44
7.0e. Three screen dumps of scope.exe demonstrating the change of zero point	Page 44
7.0f. Three screen dumps of scope.exe demonstrating the change of waveform offset	Page 44
7.0g. Six screen dumps of scope.exe demonstrating different trigger settings	Page 45
7.0h. Two screen dumps of scope.exe demonstrating different display settings	Page 46
7.0i. [Sample] tab of controls dialog	Page 46
7.0j. Grid configuration dialog box	Page 47
7.0k. Screen dump of scop.exe demonstrating multiple scope views	Page 47
7.0l. Screen dump of about dialog	Page 48
7.0m. Screen dump of configuration dialog	Page 48
7.0n. Screen dump of debug menu	Page 48
7.0o. Screen dump of advanced sampling control panel	Page 48
7.2a. Grid layout	Page 50
7.5a. Screen dump of the configuration dialog box	Page 60
7.5b. Screen dump of windows registry	Page 60
7.8a. Scope data saved to File1.sdf (Scope V1.040a, 07/04/2002)	Page 65
7.8b. Windows standard CFileDialog Box	Page 66
7.8c. Example :File1.sdf has been opened	Page 69
7.8d. Demonstrating aliasing by selecting a out of range time-base	Page 69
7.9a. Screen dump demonstrating how to copy the scope display to the clipboard	Page 71
7.9b. Scope display was pasted into this document from the clipboard	Page 71
7.10a. User clicks [Export]	Page 73
7.10b. Save dialogue box appears	Page 73
7.10c. Screen dump of exported file scope.bmp	Page 73
7.11a. Microsoft Word XP file menu	Page 77
7.11. Scope Ver1.046a file menu	Page 77

8.0. The simulator Program

Pages 86 to 89

8.0a. Screen dump of sim.exe V1.007a – 29/03/2002	Page 86
8.0b. Screen dump of options dialog in sim.exe	Page 87
8.0c. Screen dump of about dialog in sim.exe	Page 87

10.0. PIC Software Development

Pages 96 to 113

10.1a. Mark 1 flowchart	Page 96
10.2a. Mark 2 flowchart	Page 97
10.3a. Mark 3 flowchart	Page 98
10.7a. Mark 7 flowchart	Page 102
10.8a. Mark 8 flowchart	Page 104
10.9a. Mark 9 flowchart	Page 106

11.0. Test Results

Pages 114 to 147

11.1a. Test circuit diagram 1	Page 114
11.1.3a. Screen dump of scope program (sample rate = 100Hz, input wave = 1Hz)	Page 116
11.1.3b. Screen dump of scope program (sample rate = 100Hz, input wave = 2Hz)	Page 116
11.1.3c. Screen dump of scope program (sample rate = 100Hz, input wave = 5Hz)	Page 117
11.1.3d. Screen dump of scope program (sample rate = 100Hz, input wave = 10Hz)	Page 117
11.1.3e. Screen dump of scope program (sample rate = 100Hz, input wave = 20Hz)	Page 118
11.1.3f. Screen dump of scope program (sample rate = 100Hz, input wave = 50Hz)	Page 118

11.1.3g. Screen dump of scope program (sample rate = 100Hz, input wave = 100Hz)	Page 119
11.1.3h. Screen dump of scope program (sample rate = 100Hz, input wave = 200Hz)	Page 119
11.2a. Test circuit diagram 2	Page 120
11.2.3a. Screen dump of scope program (sample rate = 1,000Hz, input wave = 2.5Hz)	Page 121
11.2.3b. Screen dump of scope program (sample rate = 1,000Hz, input wave = 5Hz)	Page 122
11.2.3c. Screen dump of scope program (sample rate = 1,000Hz, input wave = 10Hz)	Page 122
11.2.3d. Screen dump of scope program (sample rate = 1,000Hz, input wave = 20Hz)	Page 123
11.2.3e. Screen dump of scope program (sample rate = 1,000Hz, input wave = 50Hz)	Page 123
11.2.3f. Screen dump of scope program (sample rate = 1,000Hz, input wave = 100Hz)	Page 124
11.2.3g. Screen dump of scope program (sample rate = 1,000Hz, input wave = 200Hz)	Page 124
11.2.3h. Screen dump of scope program (sample rate = 1,000Hz, input wave = 500Hz)	Page 125
11.2.3.1a. Screen dump of scope program (sample rate = 2,000Hz, input wave = 10Hz)	Page 125
11.2.3.1b. Screen dump of scope program (sample rate = 2,000Hz, input wave = 20Hz)	Page 126
11.2.3.1c. Screen dump of scope program (sample rate = 2,000Hz, input wave = 50Hz)	Page 126
11.2.3.1d. Screen dump of scope program (sample rate = 2,000Hz, input wave = 100Hz)	Page 127
11.2.3.1f. Screen dump of scope program (sample rate = 2,000Hz, input wave = 200Hz)	Page 127
11.2.3.1g. Screen dump of scope program (sample rate = 2,000Hz, input wave = 500Hz)	Page 128
11.2.3.1h. Screen dump of scope program (sample rate = 2,000Hz, input wave = 1,000Hz)	Page 128
11.2.3.1i. Screen dump of scope program (sample rate = 2,000Hz, trianglewave)	Page 129
11.2.3.1j. Screen dump of scope program (sample rate = 2,000Hz, squarewave)	Page 129
11.2.3.2a. Screen dump of scope program (sample rate = 5,000Hz, input wave = 50Hz)	Page 130
11.3.1a. Screen dump of scope program (Dual Channel Test)	Page 132
11.3.3a. Test Circuit Diagram 3	Page 146

1.0. PROJECT BACKGROUND

Oscilloscopes traditionally are hardware based using a CRT (Cathode Ray Tube) designed to display voltage variations (periodic or otherwise); they are bulky, expensive and have difficulty displaying low frequency waveforms.



Figure 1.0a. 20MHz Analogue Dual Trace (CRT), about £400 [W2]



Figure 1.0b. 150MHz Analogue / Digital (CRT), about £1,600 [W1]

“The word ‘Oscilloscope’ is an etymological hybrid. The first part derives from the Latin ‘oscillare’, to swing backwards and forwards; this in turn is from, ‘oscillum’, a little mask of Bacchus hung from the trees, especially in vineyards, and thus easily moved by the wind. The second part comes from the Classical Greek ‘skopein’, to observe, aim at, examine, from which developed the Latin ending ‘scopium’, which has been used to form names for instruments that enable the eye or ear to make observations.” [B1].

The heart of the traditionally CRT oscilloscope is the display screen itself, the CRT. *“The CRT is a glass bulb which has had the air removed and then been sealed with a vacuum inside. At the front is a flat glass screen which is coated inside with a phosphor material. This phosphor will glow when struck by the fast moving electrons and produce light, emitted from the front and forming the spot and hence the trace. The rear of the CRT contains the electron ‘gun’ assembly. A small heater element is contained within a cylinder of metal called the cathode. When the heater is activated by applying a voltage across it, the cathode temperature rises and it then emits a stream of electrons.” [B2].*

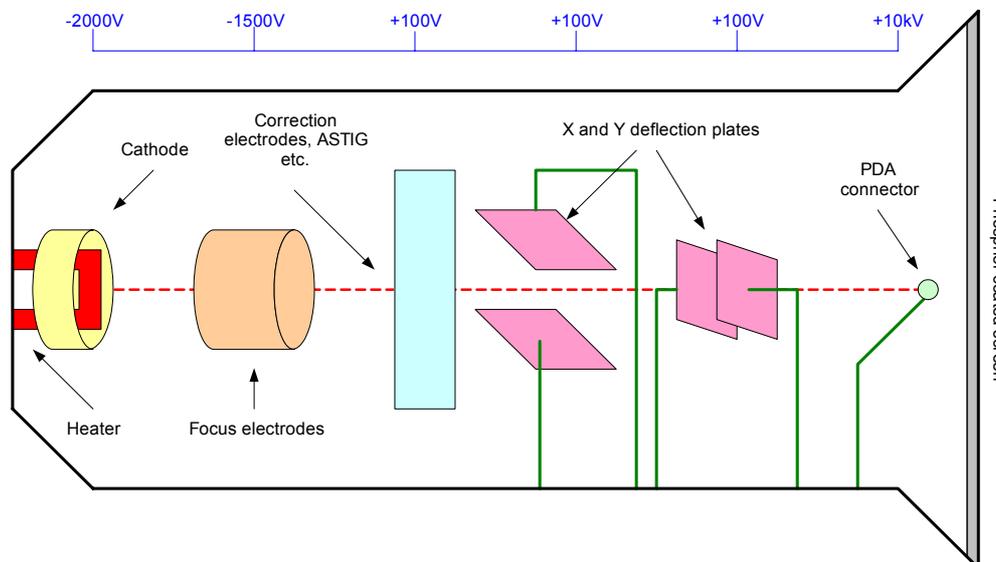


Figure 1.0c. Diagram of a typical Cathode-ray tube (CRT) construction.

This project attempts to achieve the same functionality as a traditional oscilloscope, using a PIC microcontroller for data acquisition (including appropriate analogue circuitry) which transfers the data to the PC (possibly via RS232, USB or Parallel). A Microsoft Windows based software application will then display the waveform as it would appear on a traditional CRT oscilloscope. This software application will have additional features not present on a traditional oscilloscope (e.g. printing / saving waveforms) with greater flexibility as additional features can be added as they are developed without the need for new hardware.

The digital based oscilloscope should display very low frequency waveforms in real-time, but for higher frequency waveforms it is necessary to read a finite number of samples storing them into RAM. Once the memory is full (or the preset number of samples has been reached) the PIC will stop sampling and transfer the data to the PC, when ACK (acknowledgment) is received from the PC the PIC will start sampling again. This is known as a “Storage Oscilloscope”, but there are disadvantages e.g. it’s impossible to continuously monitor a waveform in real-time for more than the amount of samples that can be stored into the buffer as there would be gaps in the data.

Digital storage oscilloscopes have two main advantages over traditional analogue scopes: -

1. The ability to observe slow and very slow signals as a solid presentation on the screen. *“Slow moving signals in the 10-100 Hz range are difficult to see and measure on a normal analogue oscilloscope due to the flicker of the trace and the short persistence of the spot on the screen. Very slow moving signals, less than 10 Hz, are impossible to view on an analogue scope. As fast as the spot traces out the waveform, the image fades and disappears before a complete picture can be formed.”* [B2].
2. The ability to hold or retain a signal in memory for long periods.

The PIC microcontroller has a built-in ADC (8, 10 or 12 bits) which has a voltage range of 0 to 5V. This voltage range is not ideal as most oscilloscopes have a much wider voltage range including negative voltages (e.g. -100 to 100V); hence an analogue circuit is required to reduce the voltage positive signals so they fall between 2.5 and 5V and voltage negative signals between 0 and 2.5V (i.e. bipolar). The built-in ADC on the PIC is slow and will limit the maximum sampling frequency; hence an external Flash ADC with direct memory access will be required to produce a high-performance digital storage oscilloscope (e.g. AD9070 – 10Bit, 100MSPS ADC).

There are commercial digital scopes, but they are expensive and have small displays (unless they have video outputs or are based on PC displays).

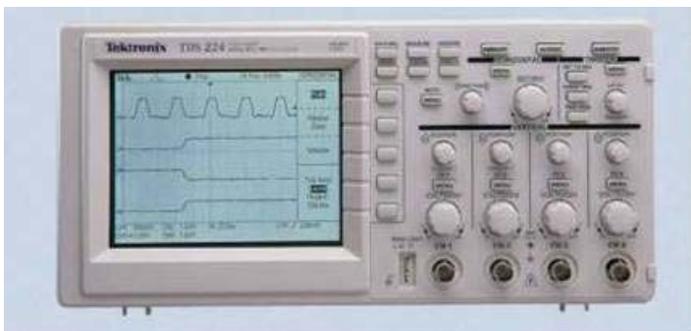


Figure 1.0d. 100MHz 4 channel digital storage oscilloscope, about £1,800. [W2]

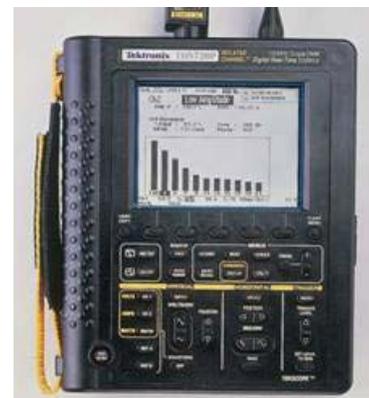


Figure 1.0e. 100MHz handheld digital storage oscilloscope, about £1,000. [W2]

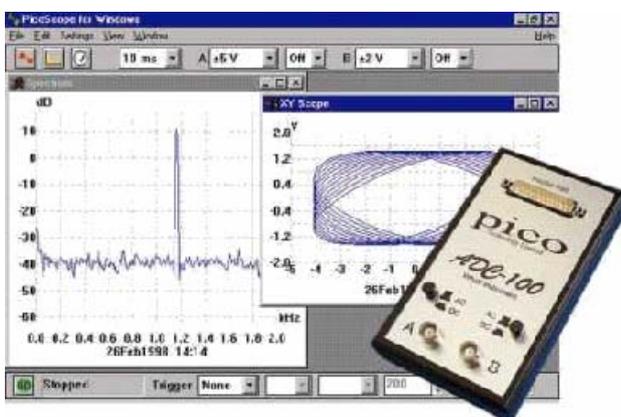


Figure 1.0f. 100kHz Dual Channel PC based storage oscilloscope (-50mV to 20V), about £250. [W2]

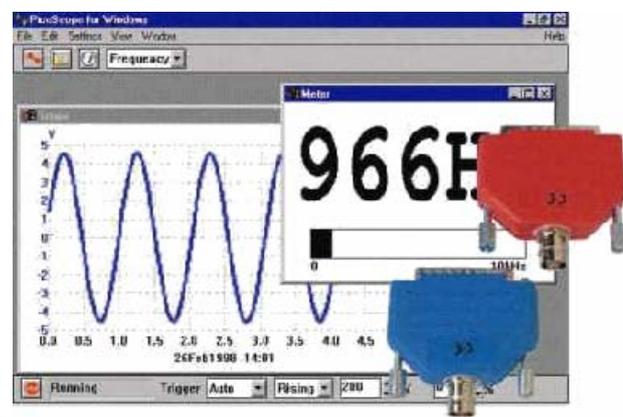


Figure 1.0g. 20kHz single channel PC based storage oscilloscope (-5 to 5V), about £95. [W2]

Advantages of the PC based oscilloscope: -

- Large screen using data projector for demonstration purposes.
- All Windows / GUI advantage such as cut & paste into documents.
- Data logging (e.g. streaming real-time data to disk).
- Remote monitoring (e.g. via the internet remotely control / view the oscilloscope from any where in the world).
- Low cost (expected to be less than £50).
- Software upgradeable.

There are many possible applications for this PC based oscilloscope: -

- Monitoring of sound waves, which are difficult to monitor on a traditional oscilloscope due to the low frequencies involved.
- Monitoring of an ECG signal, again because this is such a low frequency traditional oscilloscopes would have difficulty monitoring such a signal. ECG data could be logged and emailed directly to the doctor for diagnosis, or perhaps real-time TCP/IP internet communication so that the doctor could remotely monitor the ECG signal in real-time.
- Monitoring of serial communications, for example RS485 works on the principle of differential voltages between two cables twisted together; hence the PC based oscilloscope could be used to view serial communications. Two oscilloscope channels would be used, and the PC software will automatically add the two channels together producing a virtual trace (A+B). Note this PC based oscilloscope is also suitable for RS422 communication where there are separate transmit and receive lines (2 sets of differential twisted pair cables), hence all four scope channels can be used producing two virtual traces ($VC1 = CH1 + CH2$, $VC2 = CH3 + CH4$). This monitoring of serial communication is extremely useful for educational usage (e.g. learning how serial data is transmitted).
- The PC based oscilloscope is ideal for demonstration purposes, for example using data projector a class of student could be introduced to the oscilloscope, with real waveforms being monitored (signal generator, or even a microphone for sound waves) and displayed on a large projector display.
- Because of the low cost of the PC based oscilloscope, it is economical for a school / technical college to have large quantities available for students. Unlike traditional analogue scopes which are expensive and students are forced to share equipment, because it is not economical to purchase enough scopes for every student.

2.0. DESIGN SPECIFICATIONS

2.1. Core Objectives

1. Design and construct the hardware required for data acquisition, including analogue circuit design for insuring input voltage falls between 0 and 5V (ADC min and max range).
2. Pick a transport medium for relaying data to the PC (USB, RS232, parallel or PCI) and discuss the advantages, disadvantages and the reasons why it was picked. Design and construct the hardware required (e.g. MAX232, etc...) to interface the PIC microcontroller with the chosen medium.
3. Design a communication protocol for relaying data from the PIC to the PC in real-time (perhaps including a check-sum or CRC). Discuss the advantages and disadvantages of using a check-sum or CRC in this real-time mode.
4. Design a Windows based C++ application capable of displaying low-frequency waveforms (up-to 4 channels simultaneously <5 KHz) in real-time (i.e. the graphic subsystem). This application should directly communicate with the PIC using the chosen transport medium. The application must be user-friendly and should be Windows 95/98/NT/ME/2000/XP compatible.
5. Design PIC embedded software for reading the ADC at a certain sample rate and transferring the data to the PC in real-time.
6. A basic simulation program for simulating the PIC microcontroller, allowing the communication protocol to be tested before the hardware has been constructed. This simulation program should be able to simulate a waveform at an adjustable frequency, hence making it possible to easily test the graphic display and triggering methods in the windows based oscilloscope program.
7. Demonstrate the entire system working together as a basic real-time low-frequency oscilloscope (<5 KHz).

2.2. Further Development

1. Add storage oscilloscope functionality to the low-frequency (<5 kHz) PC based real-time oscilloscope specified in the 'core objectives' increasing it's sampling frequency to at least 20 kHz, hence making it possible to monitor sound-waves using this oscilloscope.
2. Using external ADC with direct memory access, explain how it would be possible to monitor high-frequency waveforms (storage oscilloscope >3MHz) even if the transport medium between the PIC microcontroller and the PC is much slower than the sample rate.
3. Investigate how far this storage technique can be pushed, what are the limits?

3.0 FUNDAMENTALS

3.1. The Basic CRT Oscilloscope

An oscilloscope draws its trace with a spot of light (produced by a deflectable beam of electrons) moving across the screen of its CRT (see Figure 1.0c). Basically an oscilloscope consists of the CRT, a 'time base' circuit to move the spot steadily from left to right across the screen at the appropriate time and speed, and some means (usually a 'Y' deflection amplifier) of enabling the signal to deflect the spot in the vertical or Y direction.

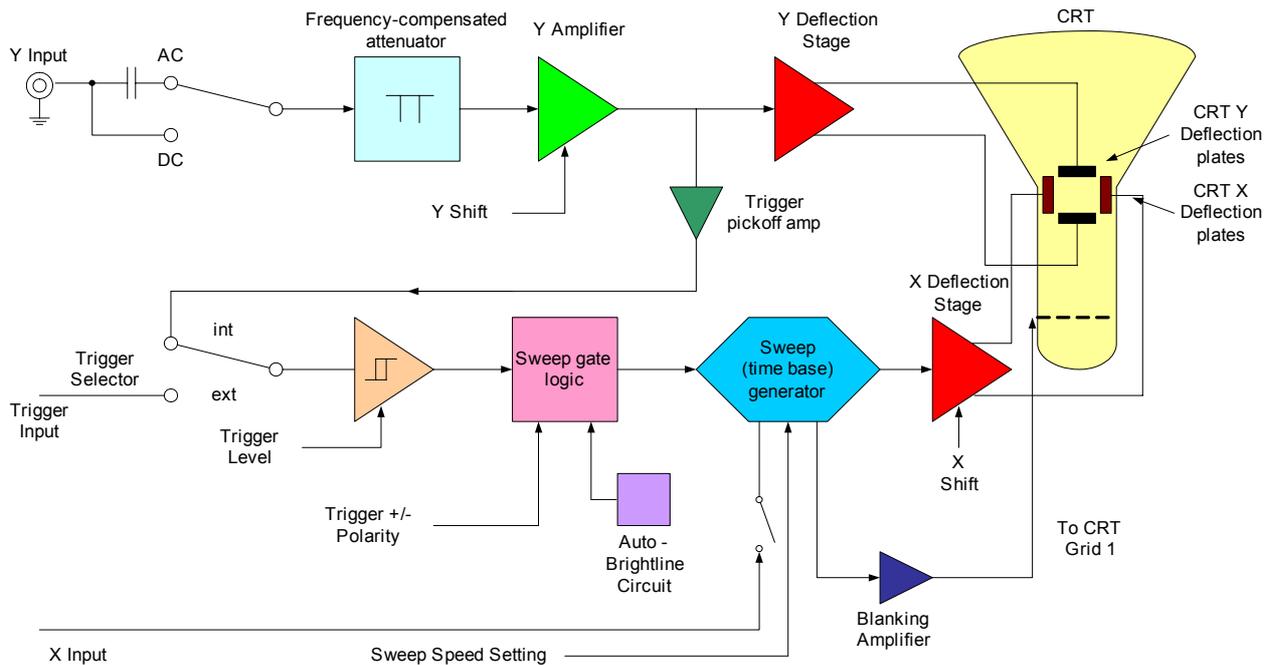


Figure 3.1a. Block diagram of a basic CRT oscilloscope

This type of oscilloscope is known as a 'real-time' oscilloscope. This means that the vertical deflection of the spot on the screen at any instant is determined by the Y input voltage at that instant. Not all CRT oscilloscope are real-time instruments, figure 3.1b categorises the various types available.

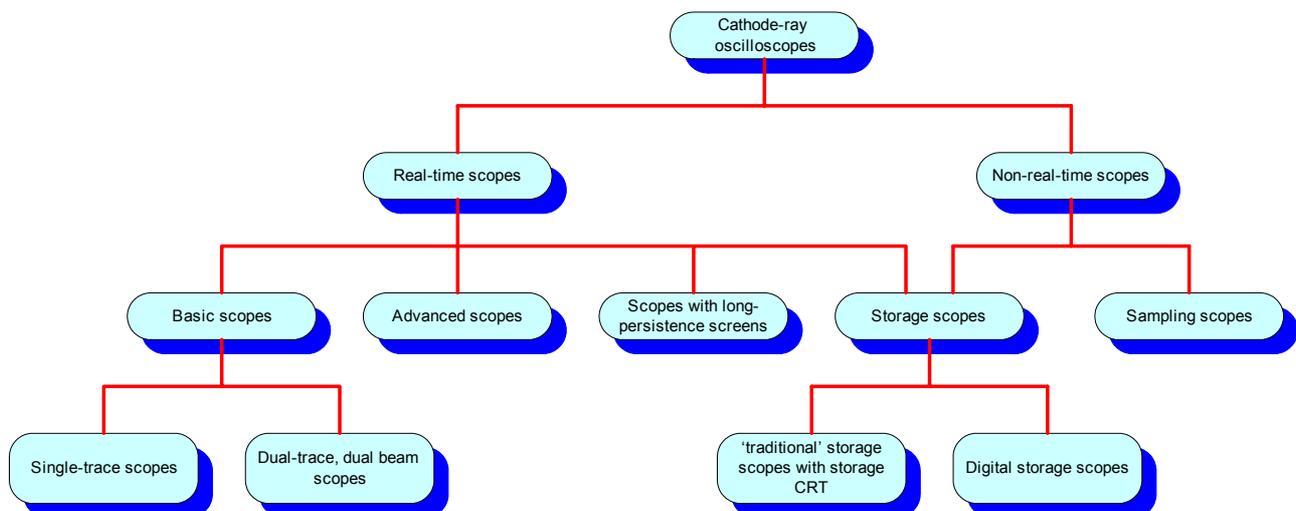


Figure 3.1b. Types of CRT oscilloscopes

3.2. Digital Sampling Oscilloscopes

Digital sampling oscilloscopes use an ADC (analogue-to-digital converter) to convert analogue voltages to binary representation. The sampling rate specifies the number of samples taken per second. Figure 3.2a demonstrates clearly how an analogue wave-form is digitally sampled and displayed onto the screen (LCD, Computer Monitor, etc...).

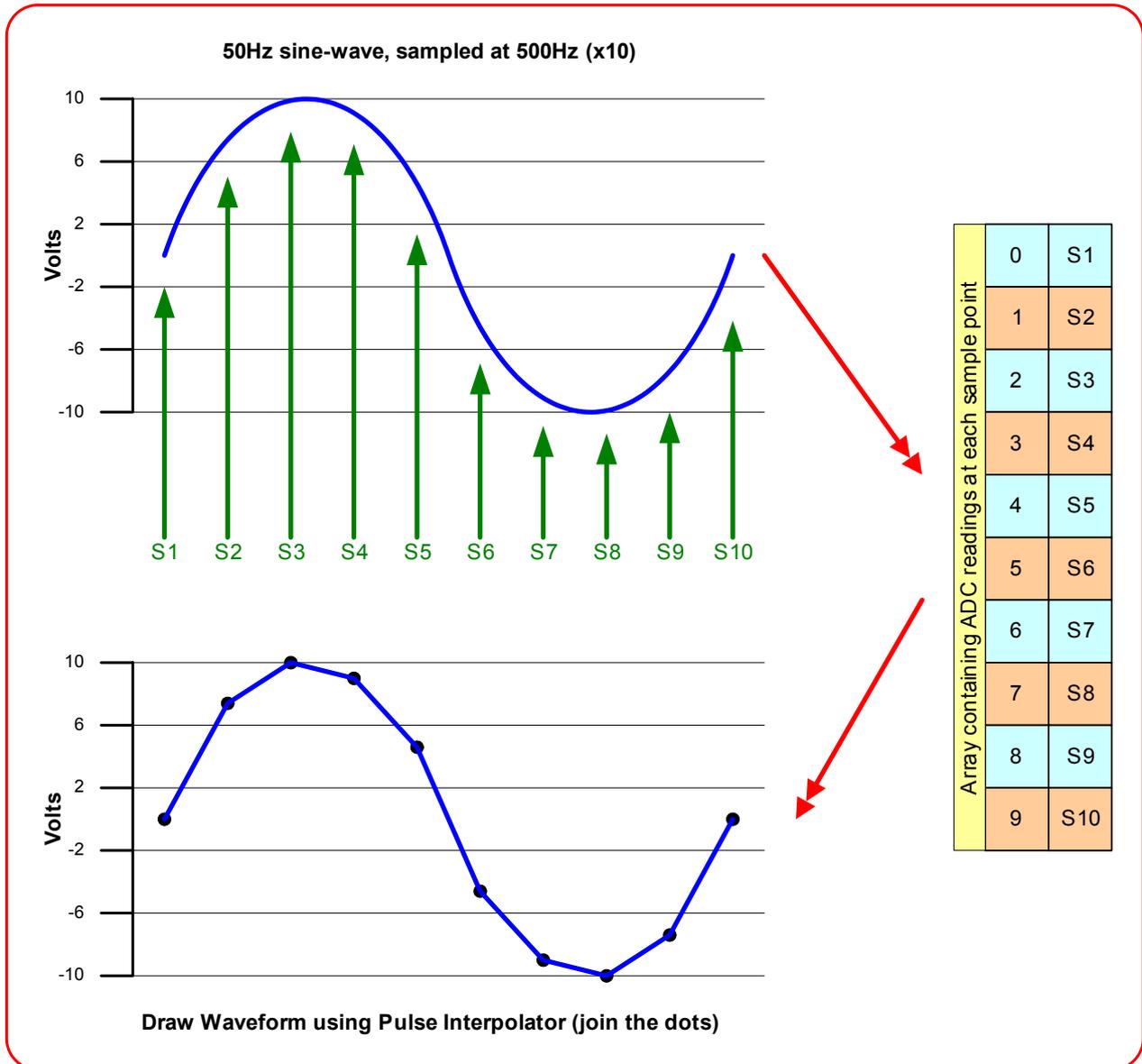
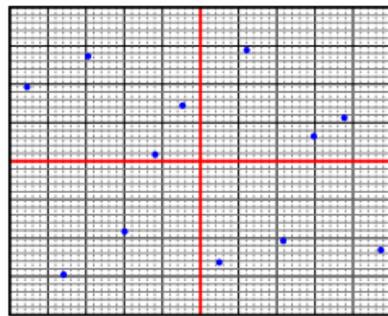


Figure 3.2a. Example showing how a sine-wave is digitally sampled

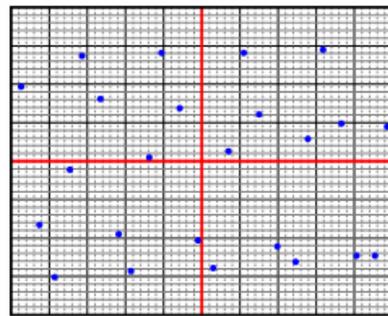
3.3. Display Formats

There are three main methods of presenting the captured waveform on the screen. These are dot display, dot joining (also called linear or pulse interpolation) and sine interpolation. These are illustrated in Figure 3.3a.

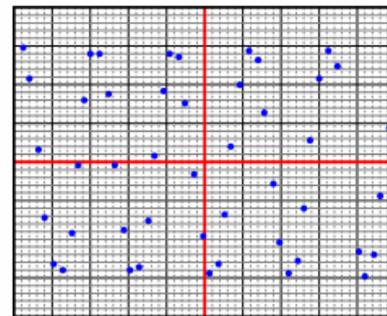
Dot display requires 20 to 25 samples per cycle before a recognisable sine wave can be plotted. Pulse interpolation provides a good general purpose display, where the waveform under investigation is known to be smooth and generally of a sinusoidal shape. Sine interpolation provides a good representation with as few as 2.5 samples per cycle. However, it should not in general be used for pulse waveforms, as here it can introduce ringing on the display which is not present on the actual waveform.



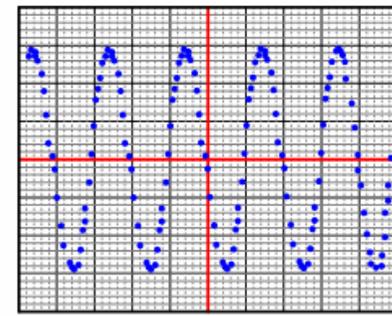
Dot Display
2.5 samples / cycle



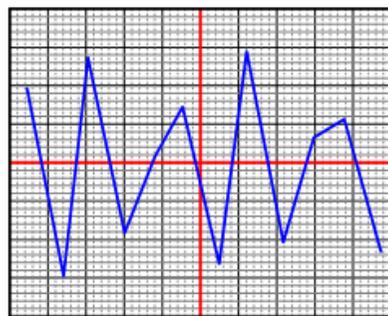
Dot Display
5 samples / cycle



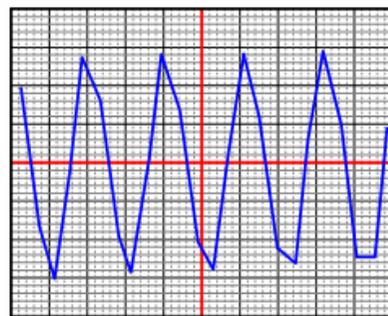
Dot Display
10 samples / cycle



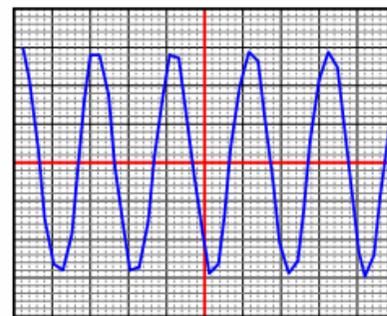
Dot Display
25 samples / cycle



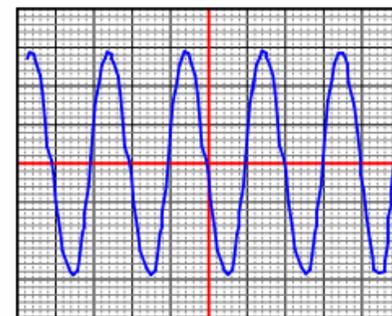
Pulse Interpolator
2.5 samples / cycle



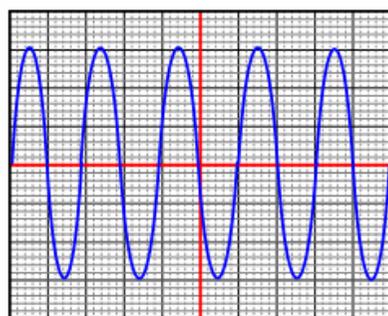
Pulse Interpolator
5 samples / cycle



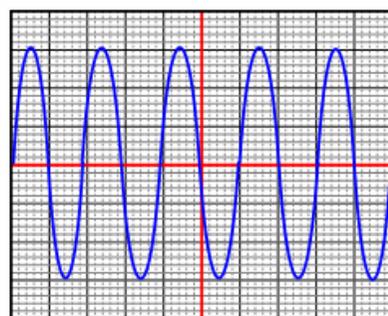
Pulse Interpolator
10 samples / cycle



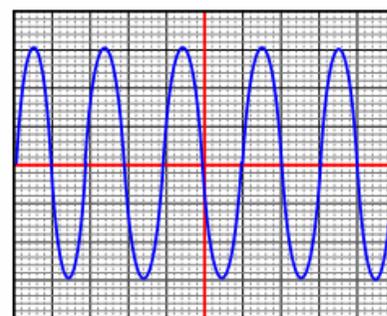
Pulse Interpolator
25 samples / cycle



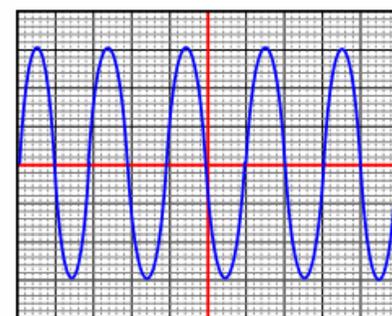
Sine Interpolator
2.5 samples / cycle



Sine Interpolator
5 samples / cycle



Sine Interpolator
10 samples / cycle



Sine Interpolator
25 samples / cycle

Figure 3.3a. Comparison of Dot, Pulse Interpolator and Sine Interpolator displays at different sample rates.

Nyquist theorem states that “to define a sine wave, a sampling system must take more than two samples per cycle”. Notice that the sine interpolator display type (2.5 samples / cycle) approaches the limits that the sampling theory suggests.

Exactly two samples per cycle (known as the ‘Nyquist rate’) suffice if it is guaranteed that they coincide with the peaks of the waveform. Otherwise there would be no knowledge of amplitude and if samples happened at zero crossings, the waveform wouldn’t even be detected. However more than two samples per cycle would be fine as the position of the samples relative to the sine wave will gradually drift through all possible phases, so that the peak amplitude will be accurately defined.

For non-sinusoidal waveforms, a sine interpolator is inappropriate, except in the case of certain instruments which can suitably pre-process the waveform before passing it to the sine interpolator.

3.4. Aliasing

Aliasing is an undesirable effect that can occur in digital sampling oscilloscopes which is not found in traditional analogue oscilloscopes. This is the display of an apparent signal which does not actually exist, usually caused by under-sampling.

Many samples should be taken per cycle (at least 10 for pulse interpolation) to ensure an accurate representation of an analogue signal in a digital memory. If only one sample is taken per cycle, or one sample per several cycles, then aliasing occurs. For example say a waveform is being sampled every three cycles, these samples may form together, particularly when using pulse interpolation, to look like a valid waveform.

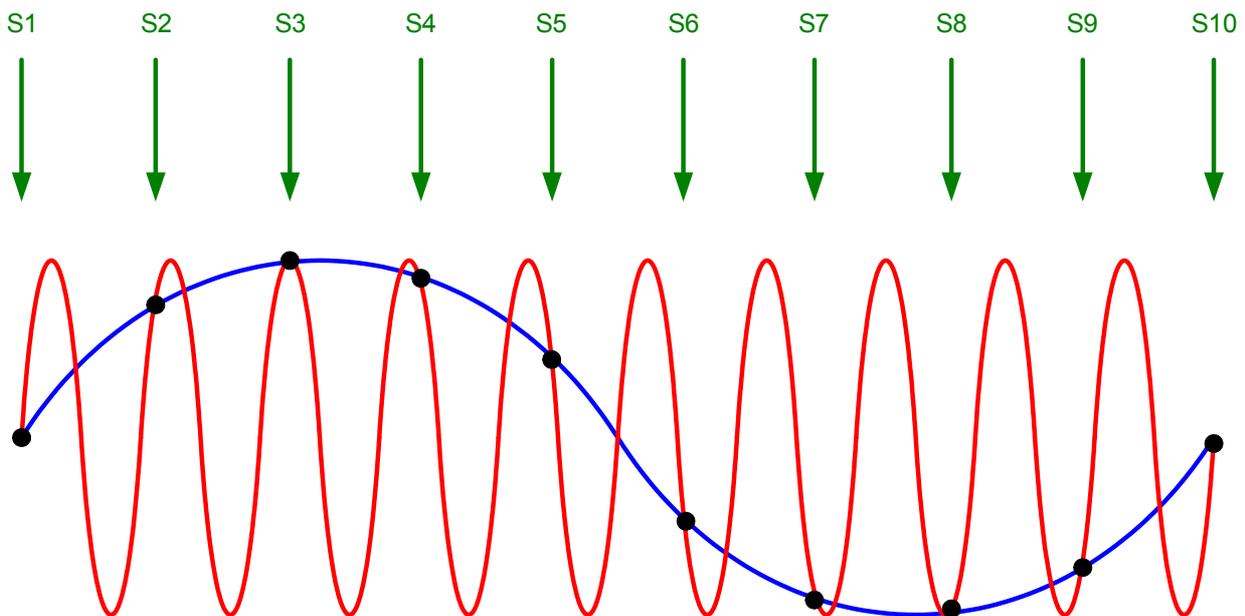


Figure 3.4a. Demonstrating aliasing, red is the real waveform, while blue is an alias.

Figure 3.4a clearly demonstrates how false signals (aliasing) can be displayed on a digital scope. The red waveform is the waveform being monitored, notice that the waveform is under sampled (see green arrows for sample points). The black dots shows where the input waveform (red) has been sampled, by joining the dots, it is clear that a perfect sine-wave is created (blue), which is an alias of the original signal. Note that it is impossible to tell that the blue signal is an alias, unless the scope has analogue and digital capabilities were a user can switch to analogue mode to check that the waveform is not an alias.

There is nothing that can be done after sampling to correct aliasing; hence the solution is to filter out high frequencies by sending the input signal through a low-pass filter. Ideally all frequencies above half the sample rate should be filtered out, but circuit design can become difficult if the sampling rate is adjustable because the high frequency cut-off must also be adjustable. One solution is to use digital potentiometers (resistance controlled by microcontroller), to change the high frequency cut-off of the filter.

3.4.1. Typical Low-Pass Filter Design Parameters

There are four key parameters, that specify an low-pass filter as shown in figure 3.4.1a: $f_{\text{CUT-OFF}}$, f_{STOP} , A_{MAX} , and M .

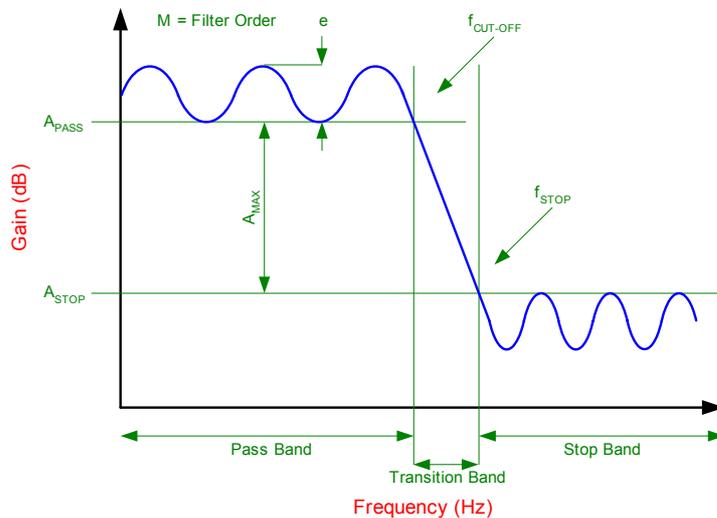


Figure 3.4.1a. The key low pass filter design parameters

The cut-off frequency ($f_{\text{CUT-OFF}}$) of a low pass filter is normally defined as the -3dB point (e.g. Butterworth and Bessel filter) or the frequency at which the filter response leaves the error band (e.g. Chebyshev).

Butterworth or Bessel filters do not create ripple in the pass band (i.e. flat) unlike the Chebyshev filter. The Chebyshev filter has a ripple up to the cut-off frequency, defined as ϵ .

“By definition, a low pass filter passes lower frequencies up to the cut-off frequency and attenuates the higher frequencies that are above the cut-off frequency.” [J4]

The filter order is determined by the number of poles in the transfer function (e.g. 3 poles, hence 3rd order). Generally, the greater number of poles a filter has the smaller the transition bandwidth.

“Ideally, a low-pass, anti-aliasing filter should perform with a ‘brick wall’ style or response, where the transition band is designed to be as small as possible. Practically speaking, this may not be the best approach for an anti-aliasing solution. With active filter design every two poles require an operational amplifier. For instance, if a 32nd order filter is designed, 16 operational amplifiers, 32 capacitors and up to 64 resistors would be required to implement the circuit. Additionally, each amplifier would contribute offset and noise errors into the pass band region of the response.” [J4]

3.4.2. Typical Low-Pass Filters

The Butterworth, Bessel, and Chebyshev are the three most popular filter designs, other filter types include: inverse Chebyshev, Elliptic and Cauer designs. See reference [B4] for detailed information on designing various types of filters including Butterworth and Chebyshev.

Butterworth Filter

The Butterworth filter is by far the most popular design used in circuits. This filter exhibits a monotonically decreasing transmission with all the transmission zeros at $\omega = \infty$, making it an all-pole filter. Therefore the transfer function of a Butterworth filter consists of all poles and no zeros and is equated to: -

$$\frac{V_{OUT}}{V_{IN}} = \frac{G}{a_0 s^n + a_1 s^{n-1} + a_2 s^{n-2} + \dots + a_{n-1} s^2 + a_n s + 1} \quad \{3.4.2.1\}$$

Note denominator coefficients for Butterworth designs are available in table form; Table 1 in reference [J4] lists all of coefficient up to a 5th order filter. The frequency behaviour has a maximally flat magnitude response in the pass-band. The rate of attenuation in the transition band is better than Bessel, but not as good as the Chebyshev filter. There is no ringing in the stop band, but there is some overshoot and ringing in the time domain, but less than the Chebyshev.

Chebyshev Filter

“The Chebyshev filter exhibits an equiripple response in the pass-band and a monotonically decreasing transmission in the stop-band. While the odd-order filter has $|T(0)| = 1$, the even-order filter exhibits its maximum magnitude deviation at $\omega = 0$. In both cases the total number of pass-band maxima and minima equals the order of the filter, N . All transmission zeros of the Chebyshev filter are at $\omega = \infty$, making it an all-pole filter.” [B4]

Therefore the transfer function of the Chebyshev filter is similar to the Butterworth filter in that it has all poles and no zeros with a transfer function of: -

$$\frac{V_{OUT}}{V_{IN}} = \frac{G}{a_0 + a_1s + a_2s^2 + \dots + a_{n-1}s^{n-1} + s_n} \quad \{3.4.2.2\}$$

Note denominator coefficients for Chebyshev designs are also available in table form; Table 2 in reference [J4] lists all of coefficient up to a 5th order filter.

Bessel Filter

The transfer function of the Bessel filter has only poles and no zeros. Where the Butterworth design is optimised for a maximally flat pass band response the transition bandwidth, the Bessel filter produces a constant time delay with respect to frequency over a large range of frequency. The transfer function for the Bessel filter is: -

$$\frac{V_{OUT}}{V_{IN}} = \frac{G}{a_0 + a_1s + a_2s^2 + \dots + a_{n-1}s^{n-1} + s_n} \quad \{3.4.2.3\}$$

Note denominator coefficients for Bessel designs are also available in table form; Table 3 in reference [J4] lists all of coefficient up to a 5th order filter. The Bessel filter has a flat magnitude response in the pass-band, and the rate of attenuation in the transition band is slower than the Chebyshev or Butterworth. This filter has the best step response of all the filters mentioned, with very little overshoot or ringing.

3.4.3. Software Generation of an Anti-Aliasing Filter

Customised filters (e.g. anti-aliasing) can be designed automatically using powerful software applications, microchip have developed a program called ‘FilterLab’. This program is freeware and can be downloaded free of charge from the microchip website [W3].

“FilterLab™ is an innovative software tool that simplifies active filter design. Available at no cost, the FilterLab active filter software design tool provides full schematic diagrams of the filter circuit with component values and displays the frequency response.

FilterLab™ allows the design of low-pass filters up to an 8th order filter with Chebyshev, Bessel or Butterworth responses from frequencies of 0.1 Hz to 10 MHz. Users can select a flat passband or sharp transition from passband to stopband. Options, such as minimum ripple factor, sharp transition and linear phase delay, are available. Once the filter response has been identified, FilterLab™ generates the frequency response and the circuit. For maximum design flexibility, changes in capacitor values can be implemented to fit the demands of the application. FilterLab™ will recalculate all values to meet the desired response, allowing real-world values to be substituted or changed as part of the design process.

Further consideration is given to designs used in conjunction with an analogue-to-digital converter. A suggested filter can be generated by simply input-ting the bit resolution and sample rate via the Anti-Aliasing Wizard. This eliminates erroneous signals folded back into the digital data due to the aliasing effect.” [W3]

For example, design a Butterworth anti-aliasing filter using the FilterLab anti-aliasing wizard. Figures 3.4.3a to figures 3.4.3f show step-by-step how to configure the anti-aliasing wizard.

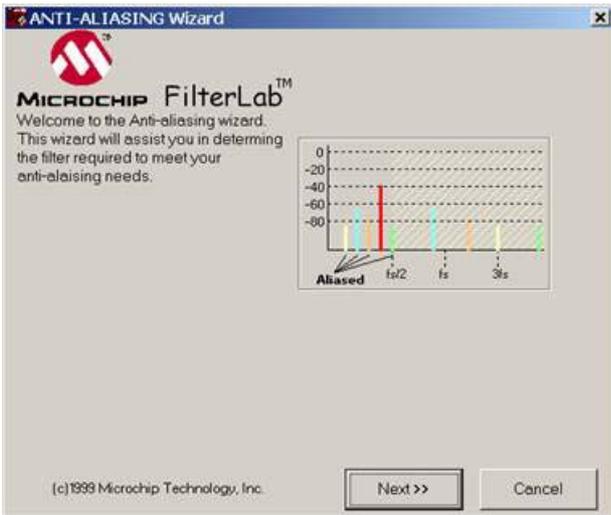


Figure 3.4.3a. Step 1: Open Anti-Aliasing Wizard

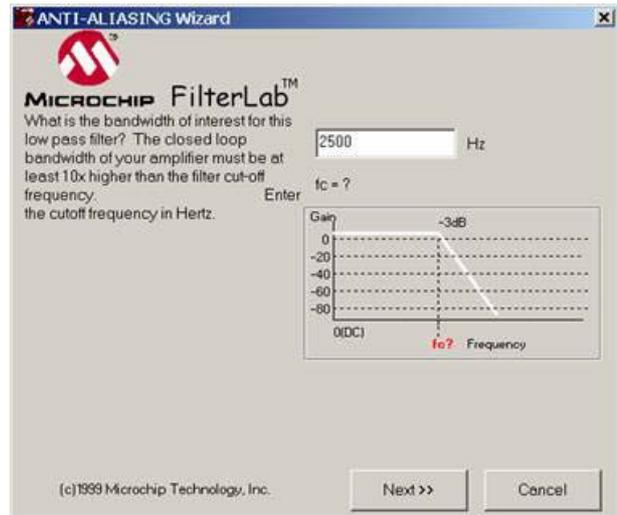


Figure 3.4.3b. Step 2: Enter 3dB Cut-off frequency

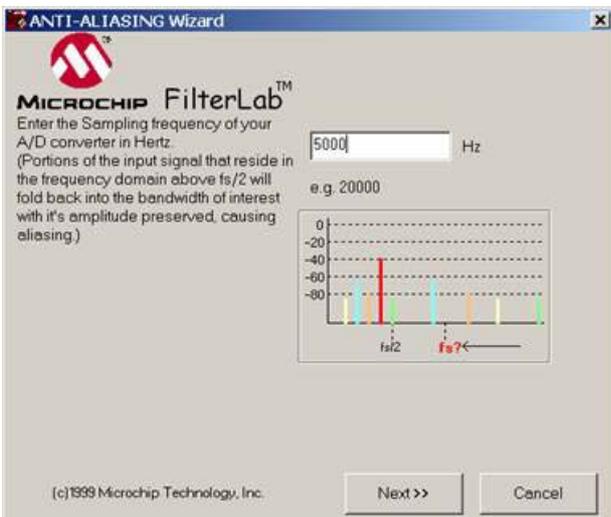


Figure 3.4.3c. Step 3: Enter sampling frequency



Figure 3.4.3d. Step 4: Enter ADC resolution

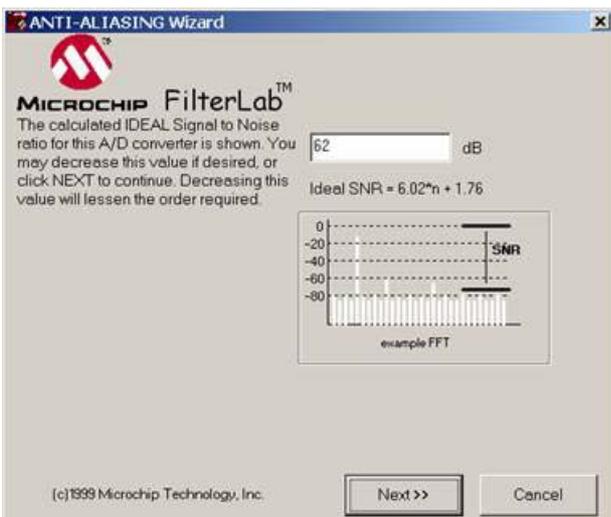


Figure 3.4.3e. Step 5: IDEAL signal to noise ratio is calculated

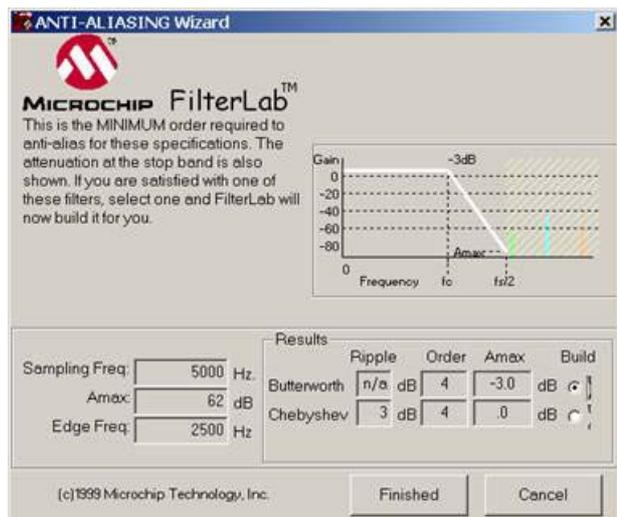


Figure 3.4.3f. Step 6: Click build Butterworth, then Finish

Notice that the anti-aliasing wizard gives the user the choice to generate a Butterworth or Chebyshev solution. Note that the Chebyshev solution (figure 3.4.3f) has a 3dB base-band ripple while the Butterworth has no base-band ripple. The anti-aliasing wizard generates the circuit diagram required to filter out the high frequencies that will cause aliasing, based on the user configurable parameters (see figure 3.4.3g).

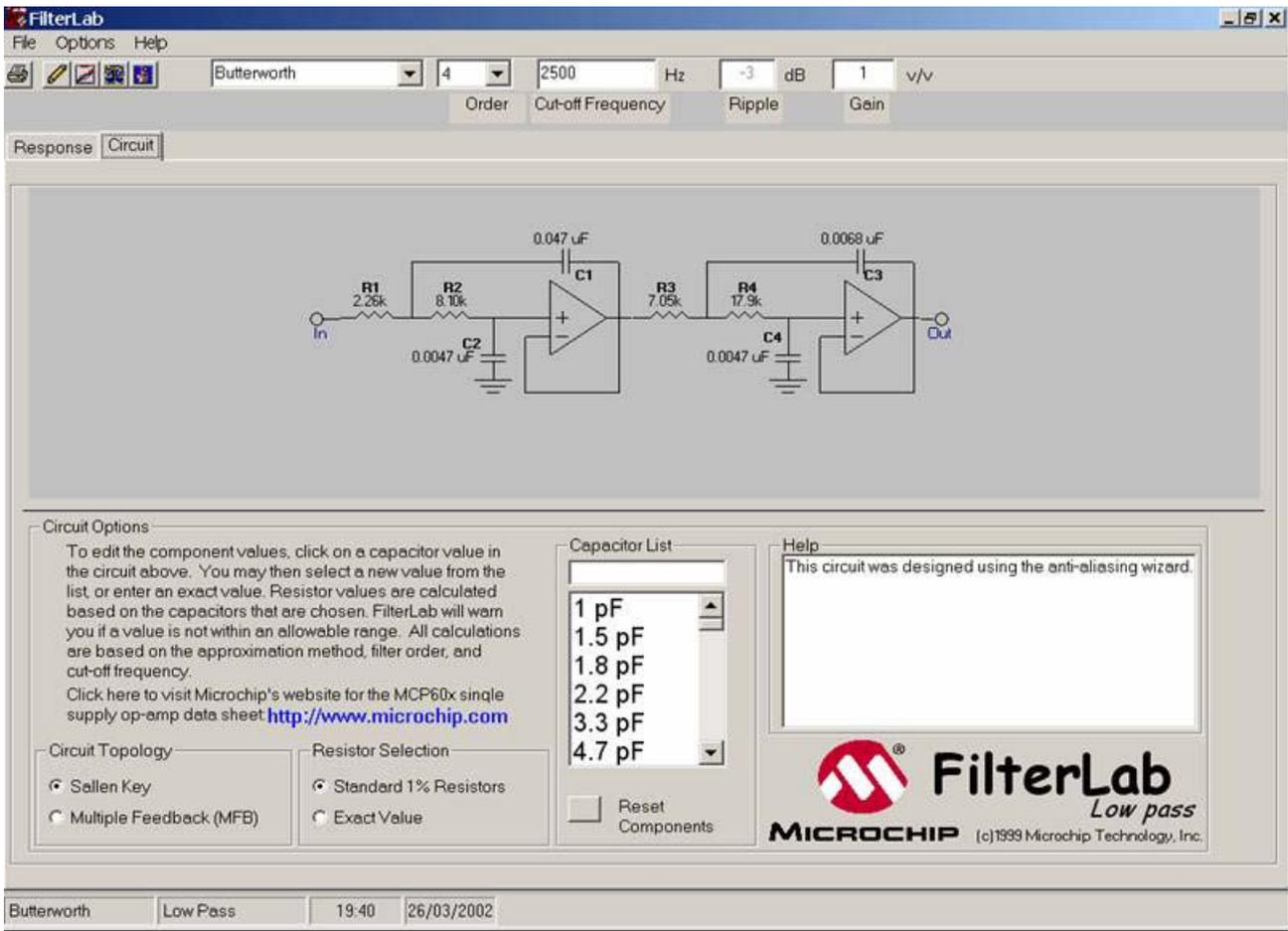


Figure 3.4.3g. Anti-aliasing filter designed by the anti-aliasing wizard

3.5. The PIC Microcontroller

A PIC (Peripheral Interface Controller) microcontroller is an IC manufactured by Microchip.



These ICs are complete computers in a single package. The only external components necessary are whatever is required by the I/O devices that are connected to the PIC.

The traditional Von-Neumann Architecture (Used in: 80X86, 8051, 6800, 68000, etc...) is illustrated in Figure 3.5a. Data and program memory share the same memory and must be the same width.

“All the elements of the von Neumann computer are wired together with the one common data highway or bus. With the CPU acting as the master controller, all information flow is back and forward along these shared wires. Although this is efficient, it does mean that only one thing can happen at any time. This phenomenon is sometimes known as the von Neumann bottleneck.” [B3]

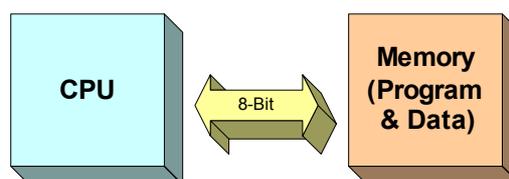


Figure 3.5a. Simplified illustration of the von Neumann architecture

PICs use the Harvard architecture. The Harvard architecture (Figure 3.5b) is an adaptation of the standard von Neumann structure with separate program and data memory: data memory is made up by a small number of 8-bit registers and program memory is 12 to 16-bits wide EPROM, FLASH or ROM.

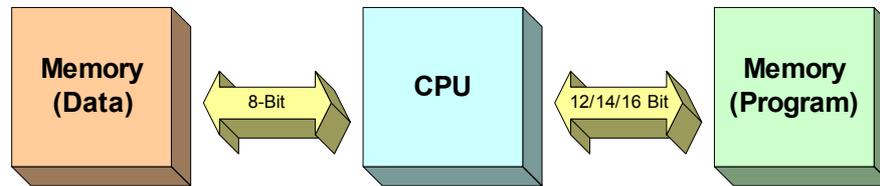


Figure 3.5b. Simplified illustration of the Harvard architecture

Traditional CISC (**C**omplex **I**nstruction **S**et **C**omputer) machines (Used in: 80X86, 8051, 6800, 68000, etc...) have many instructions (usually > 100), many addressing modes and it usually takes more than 1 internal clock cycle to execute. PIC microcontrollers are RISC (**R**educed **I**nstruction **S**et **C**omputer) machines, which have 33 (12-bit) to 58 (15-bit) instructions, reduced addressing modes (PICs have only direct and indirect), each instruction does less, but usually executes in one internal clock.

“The combination of single-word instructions, the simplified instruction decoder implicit with the RISC paradigm and the Harvard separate program and data buses gives a fast, efficient and cost effective processor implementation.” [B3]

3.5.1. Summary of the PICs Built-in Peripherals

SPI (Serial Peripheral Interface) uses 3 wires (data in, data out, clock), Master/Slave (can have multiple masters), very high speed (1.6 Mbps), and full speed simultaneous send and receive (full duplex).

I²C (Inter IC) uses 2 wires (data and clock), Master/Slave. There are lots of cheap I²C chips available; typically < 100kbps.

UART (Universal Asynchronous Receiver/Transmitter) with baud rates of 300bps to 115kbps, 8 or 9 bits, parity, start and stop bits, etc. Outputs 5V hence an RS232 level converter (e.g. MAX232) is required.

Timers, both 8 and 16 bits, many have prescalers and some have postscalers. In 14 bit cores they generate interrupts. External pins (clock in/clock out) can be used for counting events.

Ports have two control registers: TRIS sets whether each pin is an input or an output and PORT sets their output bit levels. Note: Other peripherals may steal pins, so in this respect peripheral registers control ports as well. Most pints have 25mA source/sink (LED enabled), but not all pins, it is important to look up the datasheet. Floating input pints must be tied off (or set to outputs).

ADCs (Analogue to Digital Converter) are currently slow, less than 54 KHz sampling rate (8, 10 or 12 bits), theoretically higher accuracy when PIC is in sleep mode (less digital noise) once the sample is complete the ADC sends an interrupt waking the PIC. Note that the PIC must wait until the sampling capacitor is charged; see datasheets.

3.6. RS232 Serial Interface

RS232 is simple, universal, well understood and supported, but it has some serious shortcomings as a data interface. Its origins predate modern computers and it contains many features that are not relevant to the modern user. It can control very old primitive modems and has many control signals to do this in hardware, but often it is used without these old control and status lines.

Its major feature is that it does not require the transmission of a clock, the reception of a ‘start bit’ is enough to cause the receiver to time all its actions from this one edge. This is called asynchronous transmission. RS232 allows a 5% difference in transmitted timings and receiver chip timings. This is important if using a

PIC as the datasheet specifies the % error of the baud rate generator at certain baud rates (the higher the baud rate, the higher the % error), as long as this error is less than 5% the RS232 standard is capable of coping.

Electronic data communications between elements will generally fall into two broad categories: single-ended and differential. RS232 (single-ended) was introduced in 1962, and despite rumours for its early demise, has remained widely used.

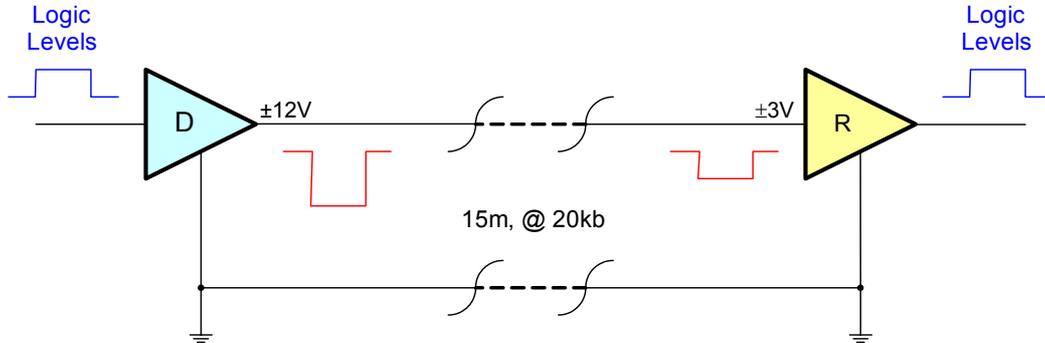


Figure 3.6a. Illustration of RS232, 1 driver and 1 receiver

“Both RS232 and RS423 are unbalanced (or single-ended) standards, where the receiver measures the potential between signal line and ground reference. Even though the transmitter and receiver grounds are usually connected through the transmission line return, the impedance over a long distance may support a significant difference in the two ground potentials, which will degrade noise immunity. Furthermore, any noise induced from the outside will affect signal lines differently from the ground return due to their dissimilar electrical characteristics – hence the name unbalanced.” [B3]

RS232 data is bi-polar, e.g. a +3 to +15 volt indicates an SPACE (ON) while a -3 to -15 volt indicates an MARK (OFF). Modern computer equipment ignores the negative level and accepts a zero voltage level as the MARK (OFF) state. This means circuits powered by 5 VDC are capable of driving RS232 circuits directly; however, the overall range that the RS232 signal may be transmitted / received is dramatically reduced.

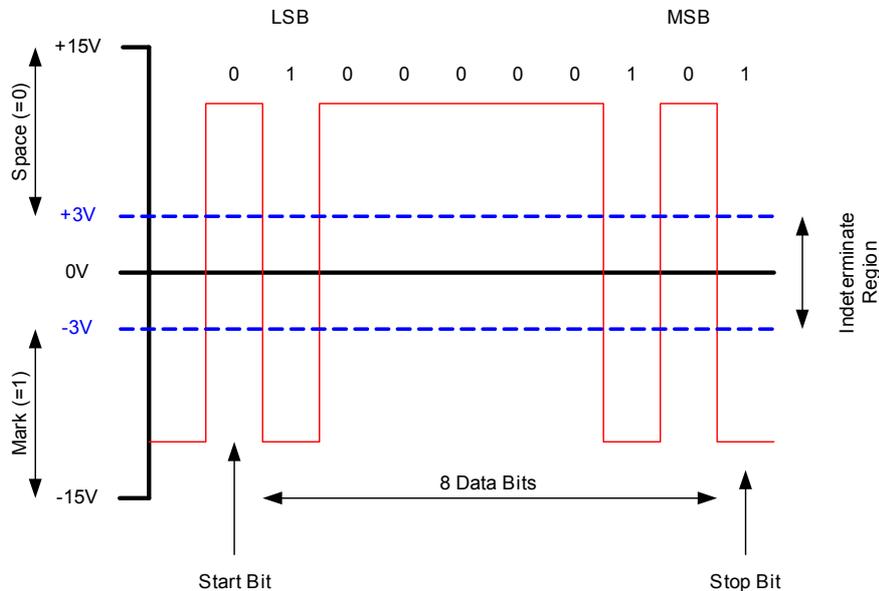


Figure 3.6d. Illustration of how data is transmitted over RS232

The output signal level usually swings between +15V and -15V. The ‘dead area’ between +3v and -3v is designed to give some noise immunity. This dead area can vary for various RS232 like definitions, for example the definition for V.10 has a noise margin from +0.3V to -0.3V. Many receivers designed for RS232 are sensitive to differentials of 1v or less.



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Receive Data	7	Request to Send
3	Transmit Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

Figure 3.6b. 9-pin RS232 D-connector, pin signal description

Typical line drivers / receivers chips for RS232 are the Maxim MAX232 or MAX233 chips (see <http://www.maxim-ic.com>) the original specification states that RS232 should drive 50 feet, but modern line driver/receivers can manage much better than this.

Baud Rate	Max Distance Shielded Cable	Max Distance Unshielded Cable
110 bps	5000 feet	3000 feet
300 bps	5000 feet	3000 feet
1200 bps	3000 feet	3000 feet
2400 bps	1000 feet	500 feet
4800 bps	1000 feet	250 feet
9600 bps	250 feet	250 feet

Figure 3.6c. Typical maximum distance modern line driver/receivers can manage before errors occur.

4.0. THE PIC16F877 MICROCONTROLLER

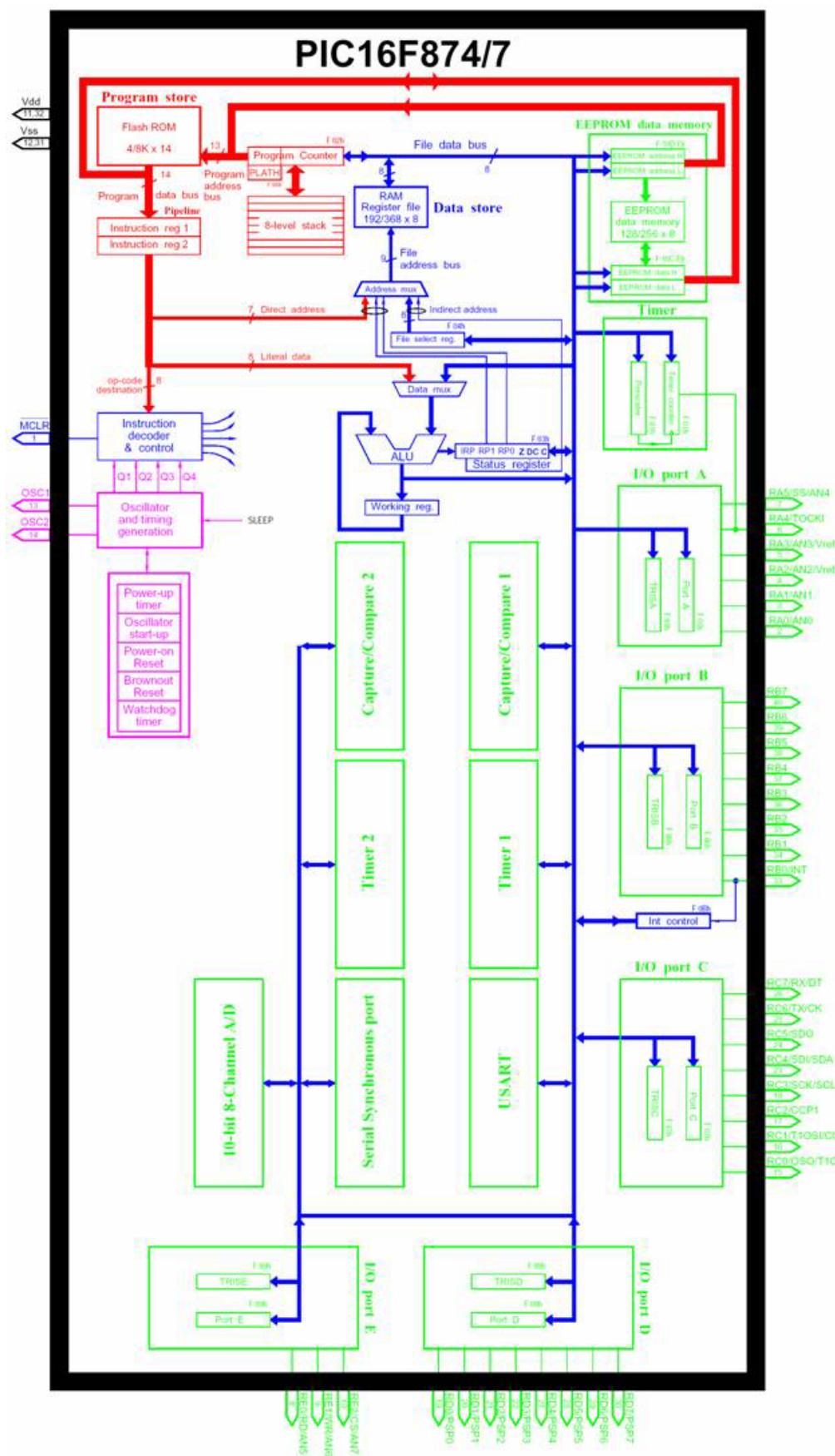


Figure 4.0a: Architecture of the PIC16F877 microcontroller [W9]

“The PIC16F877 is a high-performance FLASH microcontroller that provides engineers with the highest design flexibility possible. In addition to 8192x14 words of FLASH program memory, 256 data memory bytes, and 368 bytes of user RAM, PIC16F877 also features an *integrated 8-channel 10-bit Analogue-to-Digital converter*. Peripherals include two 8-bit timers, one 16-bit timer, a Watchdog timer, Brown-Out-Reset (BOR), In-Circuit-Serial Programming™, RS-485 type UART for multi-drop data acquisition applications, and I2C™ or SPI™ communications capability for peripheral expansion. Precision timing interfaces are accommodated through two CCP modules and two PWM modules.” [W3]

4.1. Overview of the File Registers

The data memory is partitioned into multiple banks which contain the general purpose registers and the special function registers. Bits RP1 and RP0 are the bank select bits, these bits are found in the STATUS register (b6 & b5).

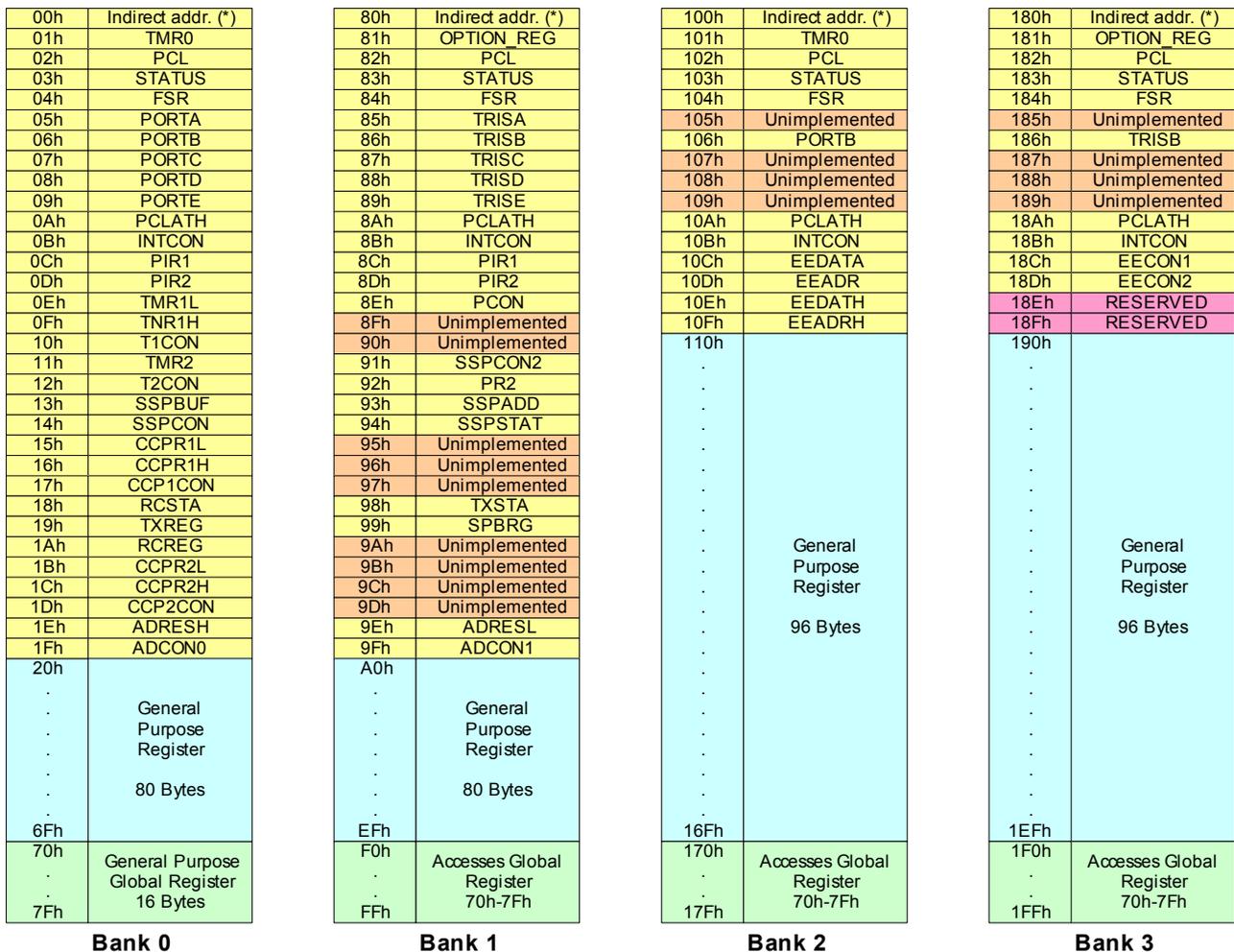


Figure 4.1a. PIC16F877 register file map

Each bank extends up to 7Fh (128 bytes). The lower locations of each bank are reserved for the special function registers (shown in yellow). Above the special function registers are general purpose registers (shown in blue), implemented as static RAM. All implemented banks contain special function registers. Some “high use” special function registers from one bank may be mirrored in another bank for code reduction and quicker access. Also notice that there are 16 general purpose global registers (shown in green), these registers can be accessed from any bank.

4.2. Overview of the 8-Channel 10-bit ADC

At first it appears that the PIC16F877 has 8 built-in ADCs, but this is not the case. Figure 4.2a shows a simplified block diagram of the analogue-to-digital converter module, clearly there is only one 10-bit ADC which can be connected to only one of eight input pins at any one time.

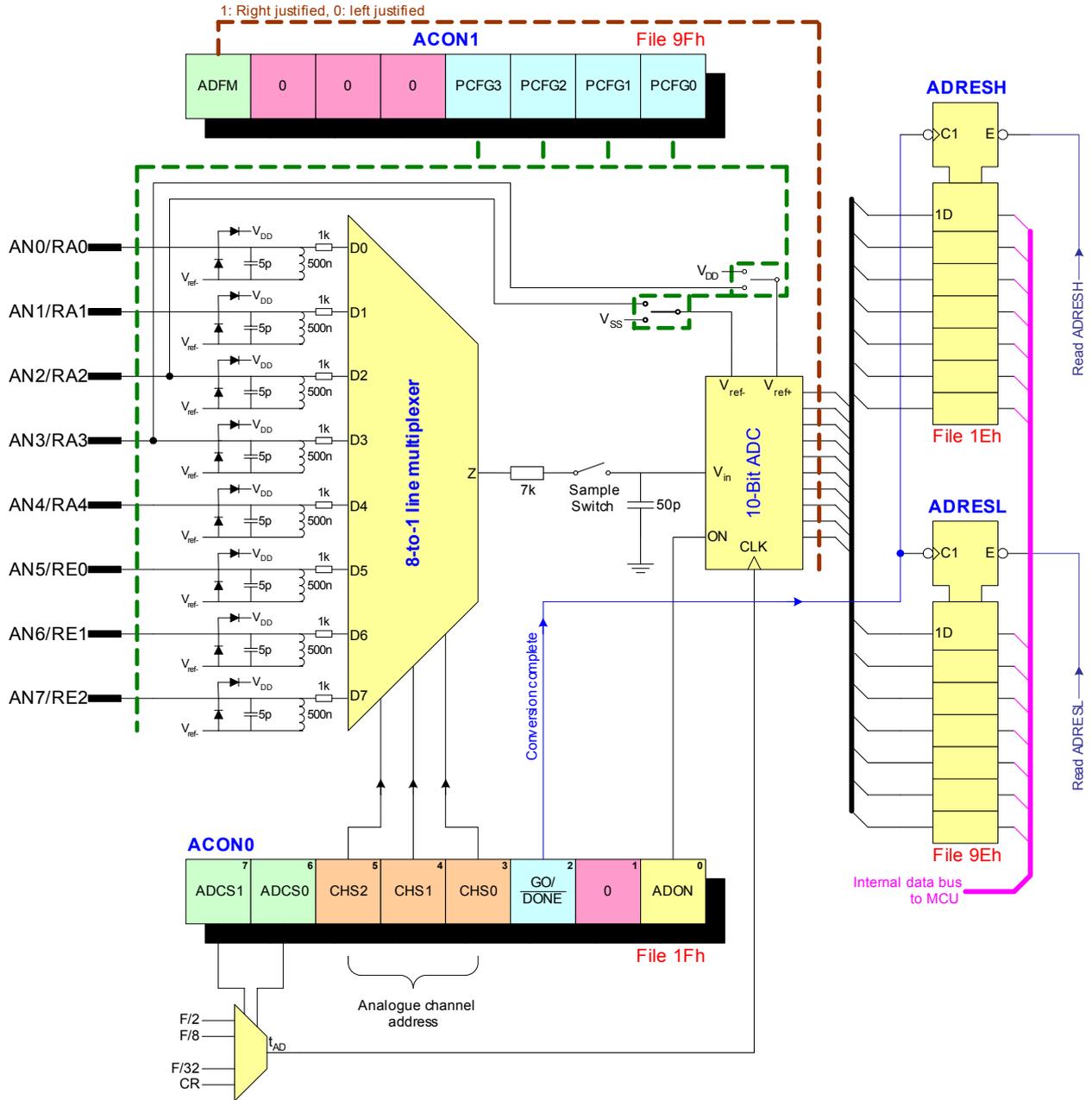


Figure 4.2a. Simplified block diagram of the PIC16F877 ADC module

The input analogue channels AN4..0 are shared with port A, and channels AN7..5 are shared with port E. If less than eight analogue channels are required then some of the pins can be assigned as digital I/O port lines using PCFG3..0 bits (see datasheet). For example, if PCFG3..0 = 0010 then AN4..0 are configured as analogue inputs, while AN7..5 are digital (port E free), with V_{DD} used as the reference.

“On reset all pins are set to accept analogue signals. Pins that are reconfigured as digital I/O should never be connected to an analogue signal. Such voltage may bias the digital input buffer into its linear range and the resulting large current could cause irreversible damage.” [B3]

The 10-bit ADC uses a technique know as ‘successive approximation’, the following mechanical analogy will help explain how it works. Suppose there is an unknown weight, a balance scale and a set of precision known weights 1, 2, 4 and 8 grams. A systemic technique can be used to calculate the unknown weight.

Place the 8g weight on the pan and remove if it is too heavy. Next place the 4g weight on the pan and remove if it is too heavy. Next place the 2g weight on the pan and remove if it is too heavy. The sum of the weights still on the pan yields the nearest lower value of the unknown weight. This is illustrated in figures 4.2b to 4.2g.

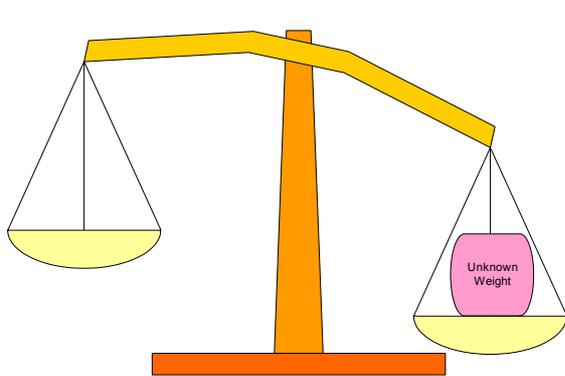


Figure 4.2b. Unknown weight placed on the scales

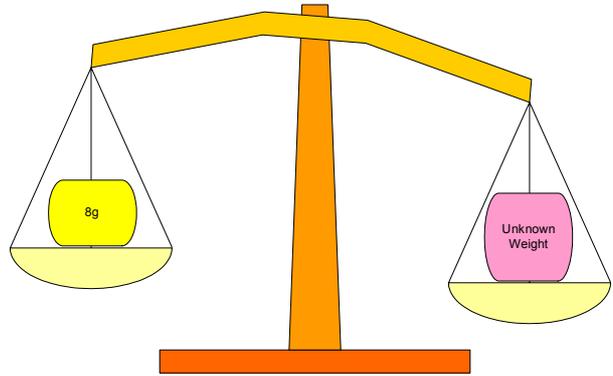


Figure 4.2c. 8g weight placed on the pan, not too heavy (keep)

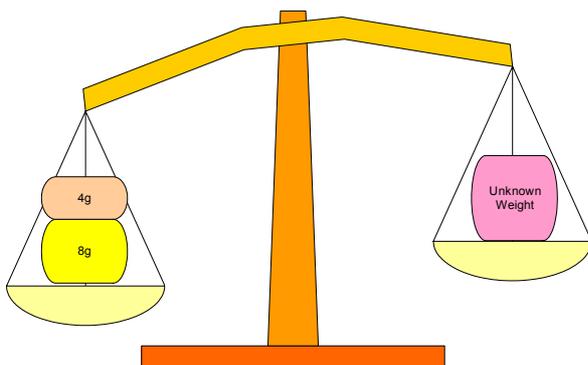


Figure 4.2d. 4g weight placed on the pan, too heavy (remove)

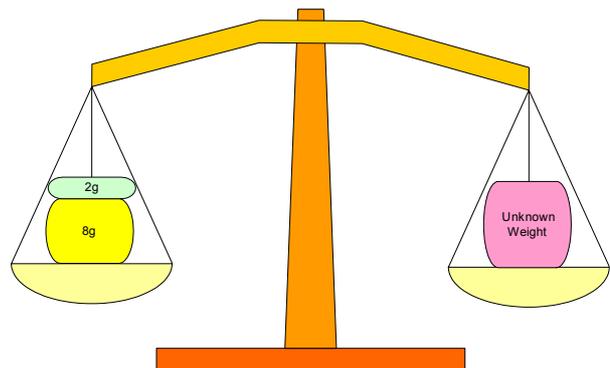


Figure 4.2e. 2g weight placed on the pan, not too heavy (keep)

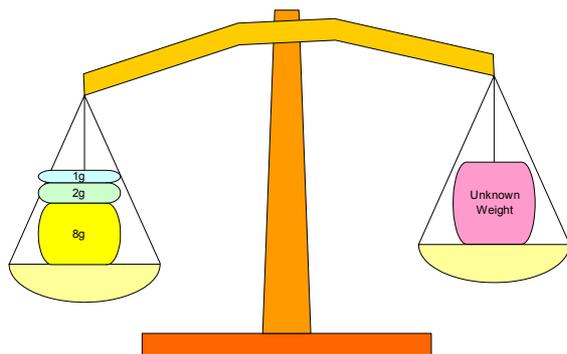


Figure 4.2f. 1g weight placed on the pan, too heavy (remove)

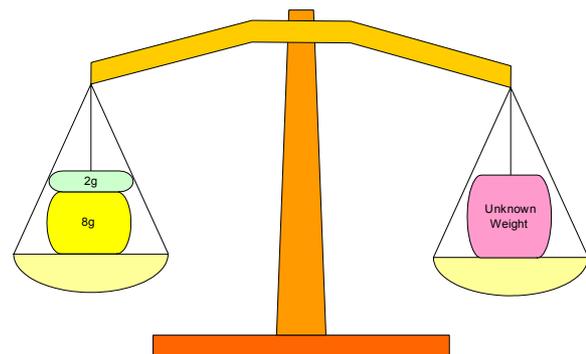


Figure 4.2g. Unknown weight is about 10g (1010)

The electronic equivalent to this successive approximation technique uses a network of precision capacitors configured to allow consecutive halving of a fixed voltage V_{REF} to be switched in to an analogue comparator, which acts as the balance scale.

Generally the network of capacitors are valued in powers of two to subdivide the analogue reference voltage (e.g. 1,2,4,8,16, etc...). This sampling acquisition process takes a finite time due to the charging time constant and is specified in the data sheet as 19.72µS.

4.3. Overview of the Hardware USART

The Universal Synchronous Asynchronous Receiver Transmitter (USART) module is one of the two serial I/O modules. The USART can be configured for asynchronous operation (UART) for communication with a PC or synchronous operation for communicating with peripheral devices such as DAC or DAC integrated circuit.

Note bit SPEN (RCSTA:7) and bits TRISC:7..6 have to be set in order to configure pin PC6/TX/CK and RC7/RX/DT for USART operation. CSS (C compiler) will automatically configure these bits, but it is important to be aware that if using fast_io(C) mode to manually configure port C, bits 7 & 6 must also be manually set if using the hardware UART.

4.3.1. Baud-Rate Generator, BRG

This is basically a programmable 8-bit counter followed by a switchable frequency ÷4 flip flop chain which can be set up to give the appropriate sampling and shifting rates for the desired baud rate, based on the PIC's crystal frequency XTAL (e.g. for 20MHz, XTAL = 20) giving: -

$$\text{Baud Rate (Low Speed Mode)} = \frac{XTAL}{64 \times (X + 1)} \quad \{4.3.1.1\}$$

$$\text{Baud Rate (High Speed Mode)} = \frac{XTAL}{16 \times (X + 1)} \quad \{4.3.1.2\}$$

$$X = \frac{XTAL \times 10^6}{64 \times BAUD} \quad \{4.3.1.3\}$$

It may be advantageous to use the high baud rate (BRGH = 1) even for slower baud clocks as this may reduce baud rate error in some cases.

5.0. HARDWARE DEVELOPMENT

The purpose of this project is to design, built and test a **low-cost** PC-based digital real-time / storage oscilloscope. The main reasoning behind hardware development was to keep the hardware cost to an absolute minimum.

It was decided to use the PIC16F877 (flash version), “The PIC16F877 is a high-performance FLASH microcontroller that provides engineers with the highest design flexibility possible. In addition to 8192x14 words of FLASH program memory, 256 data memory bytes, and 368 bytes of user RAM, PIC16F877 also features an *integrated 8-channel 10-bit Analogue-to-Digital converter*. Peripherals include two 8-bit timers, one 16-bit timer, a Watchdog timer, Brown-Out-Reset (BOR), In-Circuit-Serial Programming™, RS-485 type UART for multi-drop data acquisition applications, and I2C™ or SPI™ communications capability for peripheral expansion. Precision timing interfaces are accommodated through two CCP modules and two PWM modules.

The PIC16F877 also supports low voltage self-programming, allowing the user to program the device in-circuit at the user's operating voltage. The in-circuit debugging feature allows the designer to “emulate” the PIC16F877 device without an in-circuit emulator (the MCU itself is the “emulator”). PIC16F877 applications range from body controllers, programmable machine controls, network maintenance, feature phones and field-upgradeable pointing devices. PIC16F877 has 33 I/O pins and is available in the following package options: 40 PDIP(P), 44 PLCC(L), 44 PQFP(PQ), and 44 TQFP(PT)” [W3].

Four of the PICs ADC pins are used for data acquisition, a MAX232 buffer is used to convert the TTL serial logic of the PICs UART to the correct RS232 format. Additionally an analogue circuit is required to make sure the input voltage to the PICs ADC falls between 0 to 5V, Figure 5.0a illustrates an -10 to 10V sine wave, where the 10V peak is reduced to 5V (ADC Value is 1024), 0V is represented by 2.5V (ADC Value is 512) and -10V is represented by 0V (ADC value is 0), i.e. Bipolar. This could be achieved using two 741 op-amps, and a couple of diodes could be used for protection as the ADC cannot handle voltages outside its range (damage the PIC).

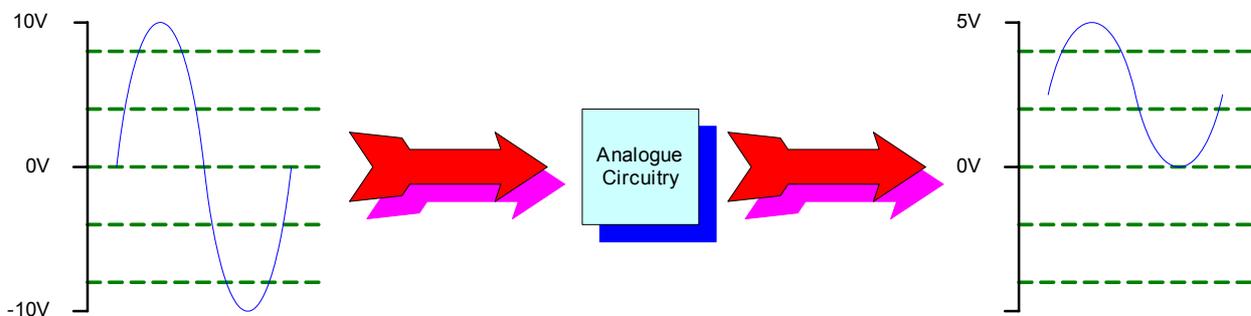


Figure 5.0a. Illustrates the operation of the analogue circuitry

A 5V voltage regulator (7805) is used to power the circuit; hence there is a large DC input range of 7 to 20V. This means that a standard DC power supply (7 to 20VDC) or a battery source (e.g. 9V PP3) can be used to power circuit.

5.1. Simplified Block Diagram

Figure 5.1a, shows a simplified block diagram of the overall system. It is clear the PIC is connected to a MAX232 buffer which is connected to the PC's RS232 port. Notice that the clock is specified as 20MHz this was not the original plan, as a slower clock speed would reduce power consumption (critical if using battery power supply) and less noise would have been generated; hence ADC readings would be more accurate, allow it is possible to put the PIC to sleep while taking the ADC reading, the ADC will send an interrupt waking the PIC once the acquisition is complete. This is not an option for this application because it takes a long time for the PIC's oscillator to return to full speed, after a sleep operation; hence this would severely affect the maximum sampling rate.

The main reason for using a 20MHz clock was because at slower clock speeds it was not possible to obtain a good 115kbps baud rate. Officially the maximum allowable baud rate error is 3%; allow most PC UARTs are specified to operate correctly with a baud rate variation of 5% before many errors start to occur. The C compiler (CCS V2.7) used to write the PIC program, generates an error if the specified baud rate cannot be achieved within 3% of the desired value using the current clock rate. The reason why 115kbps is important is because the RS232 serial communication is the bottleneck when operating in real-time mode; hence it makes sense to use the maximum baud rate possible. Note the PIC is capable at 20MHz of generating baud rates higher than 115kbps, but most PCs have a maximum baud rate of 115kbps.

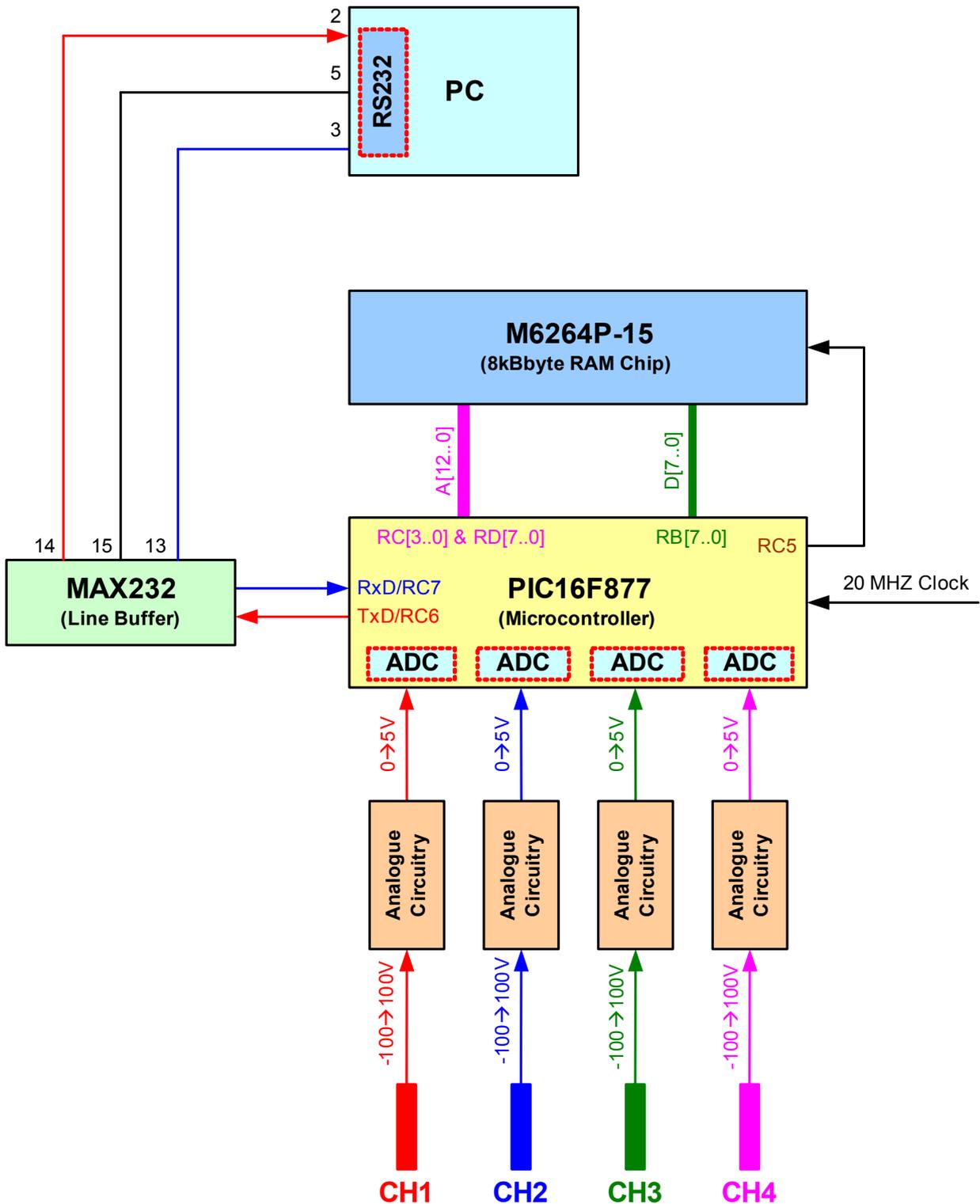


Figure 5.1a. Simplified Block diagram

The block diagram (figure 5.1a) clearly illustrates that the analogue circuitry required for each channel is identical. Note four 741 op-amps are available in a single IC chip, that's two op-amp ICs required in total (2 741s for each analogue circuit block).

Not shown on this diagram (which is one option) is the input of additional control lines from the PIC to each "Analogue Circuitry" block. These control lines will be used to select different input voltage ranges, e.g. -1V to 1V, -10V to 10V, -100V to 100V. Analogue switches could be used to switch in different resistor values to the op-amps, hence changing the gain of the op-amp (MAX4066 or MM74HC4066). A better solution is the use of digital potentiometers (resistance controlled by microcontroller), to change the gain of the op-amp. At first it appears that there is not enough free pins on the PIC to control the analogue input range, this is true, but it is possible use the data and address bus as long as the RAM chip is disabled. But this means that additional IC chips are required (latches, e.g. 74373 is an 8-bit parallel latch) to hold the selected range while the PIC is using these lines for use with the RAM chip.

The simplest way of producing different analogue ranges is to have a number of different inputs, for example each analogue block has 3 inputs: -1 to 1V, -10 to 10V and -250 to 250V. Another simple solution is to use a manual three-to-one way switch to select the appropriate range.

It is clear that a 8kByte RAM chip (see figure 5.1a) is used, this RAM chip is used during storage mode operation. A finite number of samples are stored into the RAM chip and transmitted to the PC in one large block, hence removing the RS232 bottleneck (faster sample rates possible). Notice that there are 8 data lines which are connected to port B and 13 address lines connected which are connected to port C and port D. Recall that 2 bytes are required per reading, hence 4000 readings can be stored, that's 1000 for each channel. Notice that there is one control line: this puts the RAM chip in read or write mode, but if the data lines are shared with another component sometime in the future there is the need for another control line to disable the RAM chip.

It is possible to use four PICs (one for each channel) hence each channel can be sampled simultaneously, using a master/slave communication protocol. For example each PIC has its own address (say 1 to 4) all four wait until the master (PC) calls them before transmitting their data. Note: the current communication protocol would not work because there is a high likelihood of a collision occurring (two or more transmit at the same time), hence the need for a master/slave communication protocol or extremely good synchronisation where each PIC only transmits during specified time blocks. Sampling simultaneously is not required, the two modes chop and alt should be adequate, hence only one PIC is required (chop: read CH1 → read CH2 → read CH3 → read CH4 → ..., alt: read CH1 1000 times → read CH2 1000 times → read CH3 1000 times → read CH4 1000 times). Simultaneous sampling using four PICs may be useful for specialist applications and this project is suitable as only mirror software modification is required.

5.2. Digital Circuit Diagram

Figure 5.2a shows the digital circuit diagram, the design is simple with a low chip count (3 ICs).

S2	S1	S0	Baud Rate
0	0	0	115,200 bps
0	0	1	57,600 bps
0	1	0	38,400 bps
0	1	1	32,768 bps
1	0	0	19,200 bps
1	0	1	14,400 bps
1	1	0	9,600 bps
1	1	1	4,800 bps

Figure 5.2b. Dip-switch configuration

The dip-switches connected to port E are used to select RS232 baud rate (see figure 5.2b). The MAX232CPE (RS232 line buffer) is connected to pins RC7 and RC6 of the PIC; these pins are for use with the PICs hardware UART. The use of the PICs hardware UART was the only option, because a software UART would severely affect system performance.

The reason why the MAX232CPE (RS232 line buffer) was chosen was because it can be powered from a single 5V power supply. Recall that RS232 requires +3 to +12 volts for a logic '0' and -3 to -12 volts for a logic '1', the MAX232CPE has a built-in (external capacitors required)

voltage doubler circuit (+10V) and a voltage inverter circuit (-10V). This reduces product cost as the other option is to using a switch mode DC to DC converter (cost about £5) to generate required power supply. Allow a +15 and -15 supply is required for the analogue circuit; hence a DC to DC converter is required anyway. Allow the max232 data sheet states that the +10 and -10 voltage pins could be used to drive other circuits, but it is not recommended, plus its good design practice to keep analogue and digital circuits separate (problems with noise).

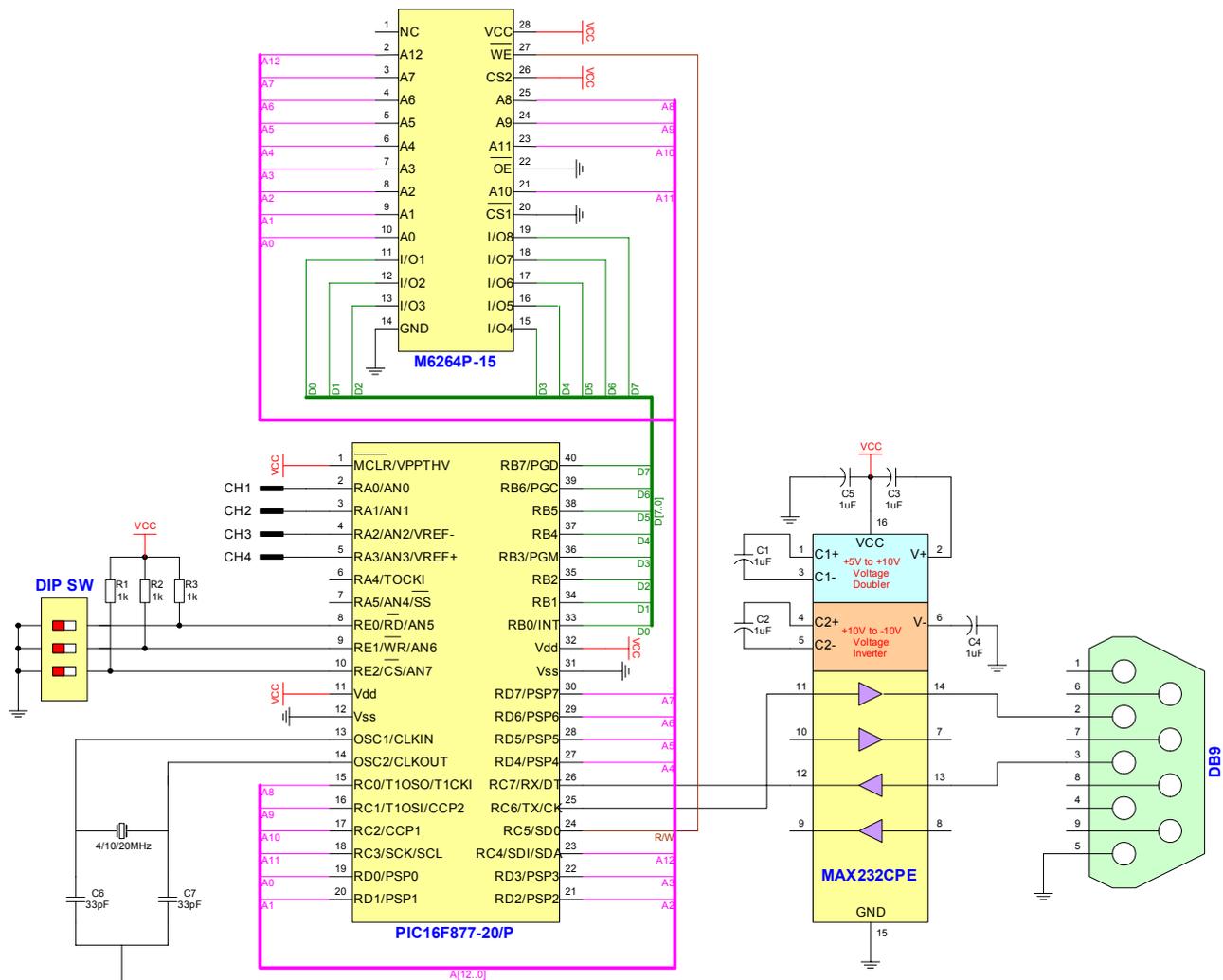


Figure 5.2a. Digital circuit diagram (version 1.3, 19/03/2002)

Port D is used for the first 8-bits of the 13-bit address bus ($2^{13} = 8192$ address), RC4 to RC0 is used for the upper 5-bits of the address bus and Port B is used for the 8-bit data bus. The only component making use of the data and address buses is the external RAM chip, hence there is only the need for one control line (R/W), this control line (R/W) puts the RAM chip in read or write mode. If the data and buses are shared with other components sometime in the future the RAM chip must be disable, while communicating with these devices, this requires an additional control line (/OE).

To reduce cost even further a 28-pin version of the chosen PIC (PIC16F876) could be used, but this means a serial RAM chip must be used. The operation of a serial RAM chip is much slower than a parallel RAM chip, as each bit of the address and data must be clocked in one bit at a time. The purpose of the RAM chip is to compensate for the RS232 bottleneck (serial link); surely a serial RAM chip would have the same bottleneck. Yes it does, but the maximum baud rates are much faster than RS232, hence a serial RAM chip is a real possibility especially now the PIC is running at 20 MHz (must used 20MHz for high RS232 baud rate).

The ADC input channels (RA3..RA0) labelled CH4..CH1 are connected to the output of the analogue circuit.

5.3. Analogue Circuit Diagram - Mark 1

Figure 5.3a shows the circuit diagram for ensuring that the input voltage falls between zero and five volts. The first op-amp is designed to change the input voltage so that it does not go over 0 volts, for example changes -1 to 1V, to 0 to -2V. The second op-amp is configured in a negative amplifier mode to increase and

change 0 to -2V, to 0 to 5V. Note the third part with the two zener diodes is for protection, the PIC ADCs may be damaged if input voltage is larger than 5 volts or smaller than 0 volts.

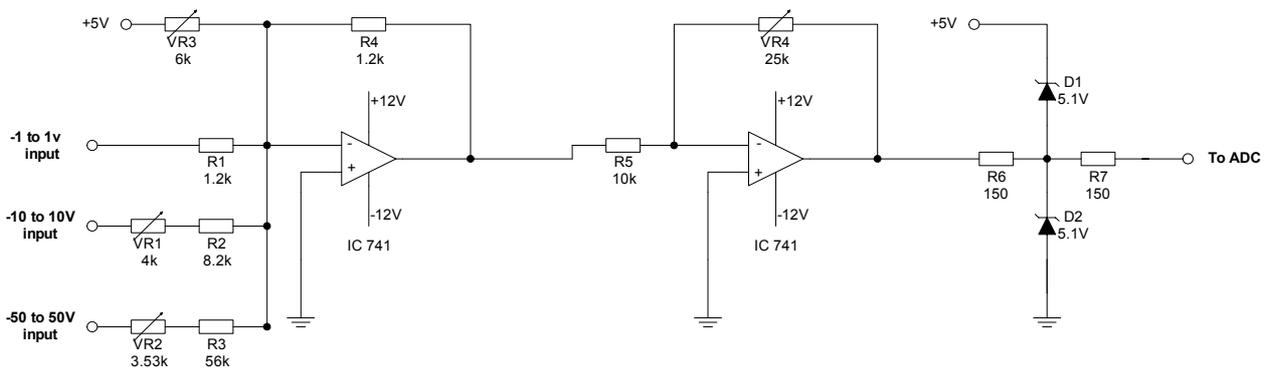


Figure 5.3a. Analogue circuit diagram – Mark 1 (14/12/2001)

The variable resistors are for calibration: -

1. Ground -1 to 1V input, calibrate using VR3 until -1.5V is outputted from the first op-amp.
2. Change VR4 until 2.5 volts is outputted from the output after the two zener diodes and the 2 150R resistors.
3. Un-ground -1 to 1v input and put in a voltage into input -10 to 10v and calibrate using VR1. Disconnect that voltage.
4. Put a voltage in to -50 to 50V and calibrate using VR2.

Disadvantages of this circuit: -

1. Use of variable resistors (much more expensive than fixed value resistors).
2. No op-amp protection if wrong input line is used (e.g. 100V input on the -1 to 1V input).
3. If an input line is used (say -10 to 10V line) and a device is connected to another line (e.g. say -1 to 1V line) there is nothing to stop current flow to the other device. A diode could be used but there is a 0.7 voltage drop across the diode, resistor values could be changed to compensate for this voltage drop.
4. Range not microcontroller controlled.

5.3.1 Simulation of the Circuit using Electronic Workbench

From figure 5.3.1a (using -1 to 1V input) it is clear that a -1 to 1V sine-wave input has been applied and the output is 0 to 5V. Figure 5.3.1b clearly shows an input of -0.5v to 0.5v, and an output of to 1.25V to 3.75V. Notice the input and output waveforms are the same phase and polarity.

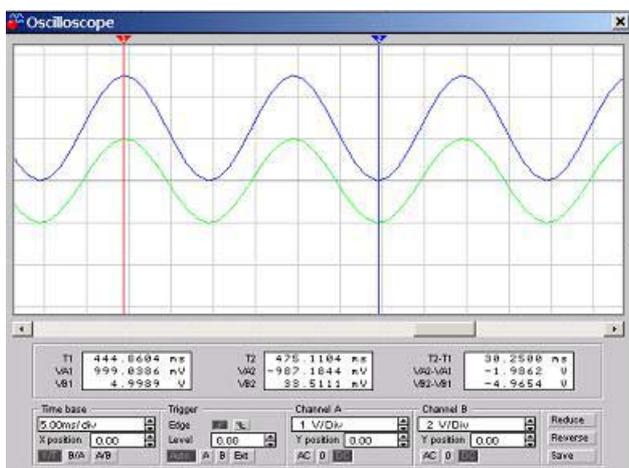


Figure 5.3.1a. Screen dump of oscilloscope (CH A: Input, CH B: Output)

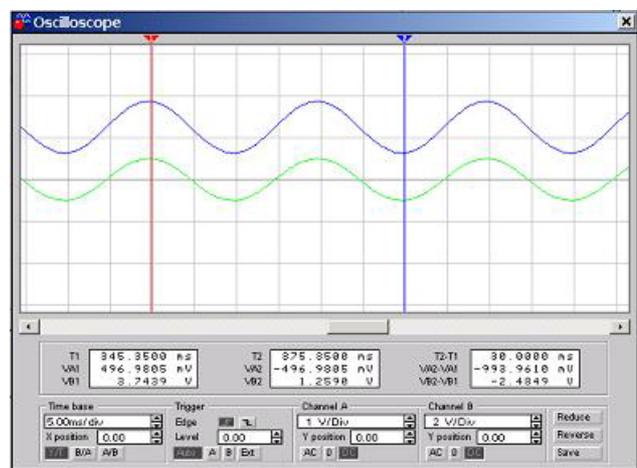


Figure 5.3.1b. Screen dump of oscilloscope (CH A: Input, CH B: Output)

From figure 5.3.1c (using -10 to 10V input) it is clear that a -10 to 10V sine-wave input has been applied and the output is 0 to 5V. Figure 5.3.1d clearly shows a triangle-wave input of -5v to 5v, and an output of to 1.3V to 3.70V. Notice the input and output waveforms are the same phase and polarity.

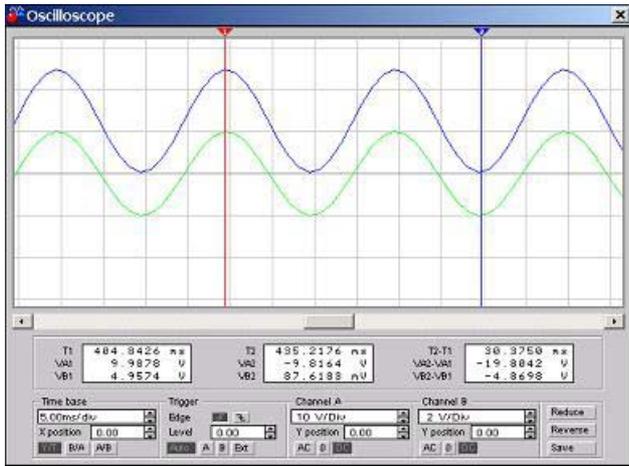


Figure 5.3.1c. Screen dump of oscilloscope (CH A: Input, CH B: Output)

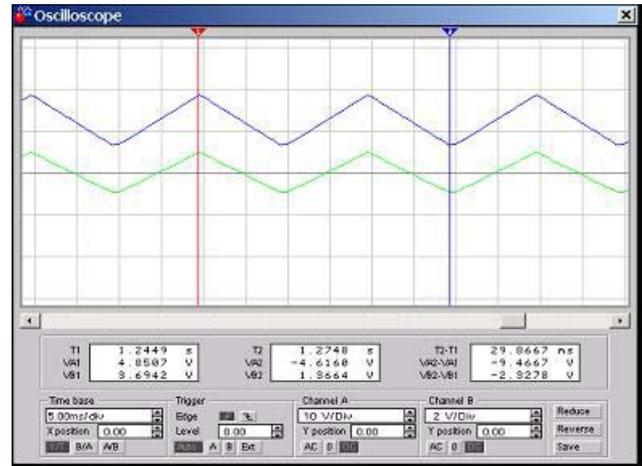


Figure 5.3.1d. Screen dump of oscilloscope (CH A: Input, CH B: Output)

From figure 5.3.1e (using -50 to 50V input) it is clear that a -50 to 50V sinewave input has been applied and the output is 0 to 5V. Figure 5.3.1f clearly shows a squarewave input of -25v to 25v, and an output of to 1.24V to 3.76V. Notice the input and output waveforms are the same phase and polarity.

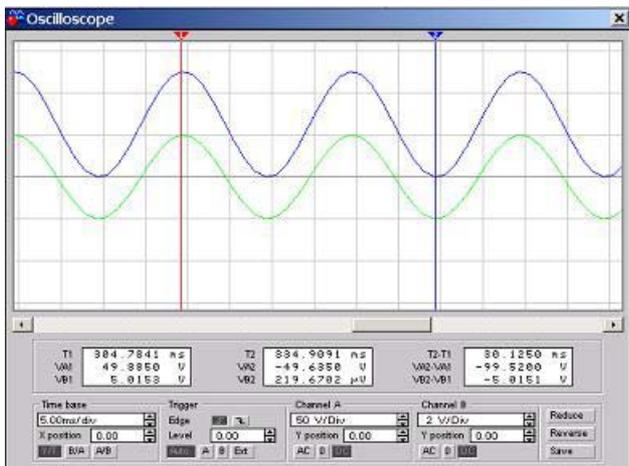


Figure 5.3.1e Screen dump of oscilloscope (CH A: Input, CH B: Output)

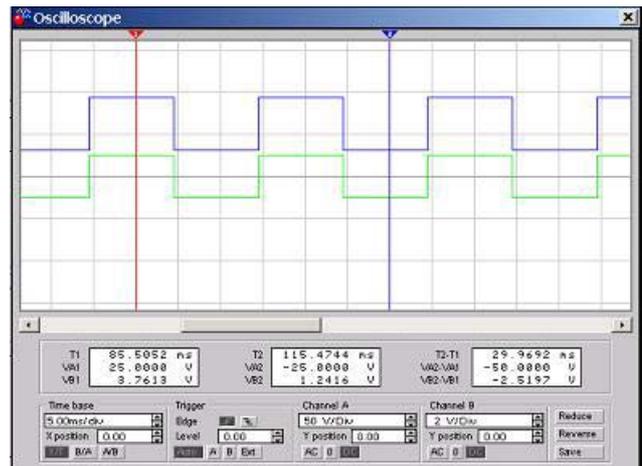


Figure 5.3.1f. Screen dump of oscilloscope (CH A: Input, CH B: Output)

5.4. Analogue Circuit Diagram - Mark 2

Figure 5.4a shows the modified circuit diagram for ensuring that the input voltage falls between zero and five volts, without the use of variable resistors. The first op-amp is designed to change the input voltage so that it does not go over zero volts, for example changes -2.5 to 2.5V, to -5 to 0V. The second op-amp is configured in negative amplifier mode with a gain of unity changing -5 to 0V to 0 to 5V. The third part with the two diodes is for protection, e.g. say a -1 to 6V signal makes it through the op-amps the diodes will cut off the peaks of the waveform making sure no damage is done to the PICs ADC. Note for the -250 to 250V range it may be necessary for R3 to be made up of several resistors in series as standard resistors have a maximum operating voltage of about 200 volts. All resistors should have a tolerance of at least 1%; ideally instrumental resistors should be used.

Instead of three inputs it is recommend that a three-to-one line manual switch should be used to connect each line to a single input line. Over voltage protection is still required in case of miss use, e.g. the user has the -2.5 to 2.5V volt input selected and connects the 240V_{AC} mains to the input.

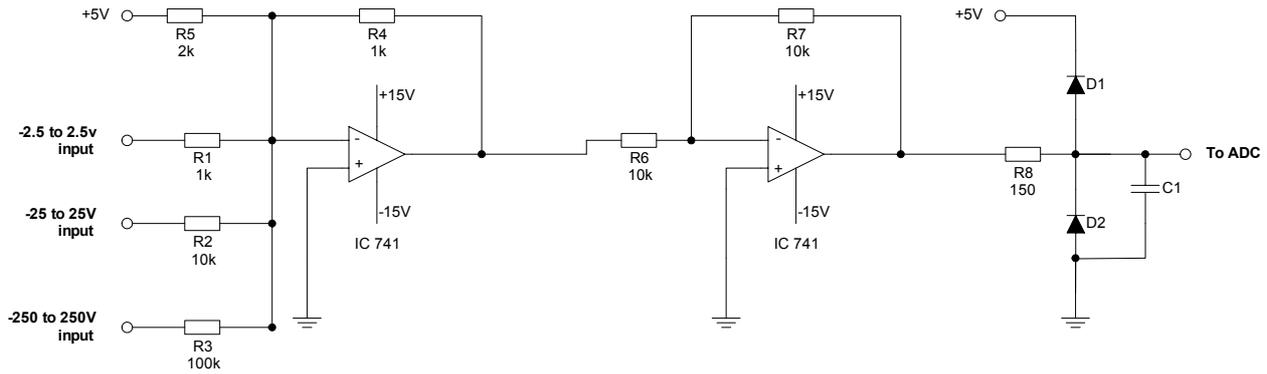


Figure 5.4a. Analogue circuit diagram – Mark 2 (19/03/2002)

Disadvantages of this circuit: -

1. No op-amp protection if wrong input line is used (e.g. 100V input on the -2.5 to 2.5V input).
2. Range not microcontroller controlled.

5.4.1 Simulation of the Circuit using Electronic Workbench

From figure 5.4.1b (using -2.5 to 2.5V input) it is clear that a -2.5 to 2.5V sine-wave input has been applied and the output is 0 to 5V. Figure 5.4.1c clearly shows an input of -1v to 1v, and an output of to 3.5V to 1.5V. Notice the input and output waveforms are the same phase and polarity.

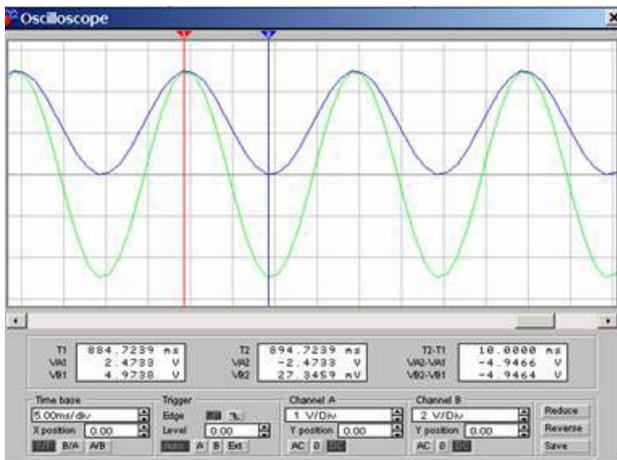


Figure 5.4.1b. Screen dump of oscilloscope (CH A: Input, CH B: Output)

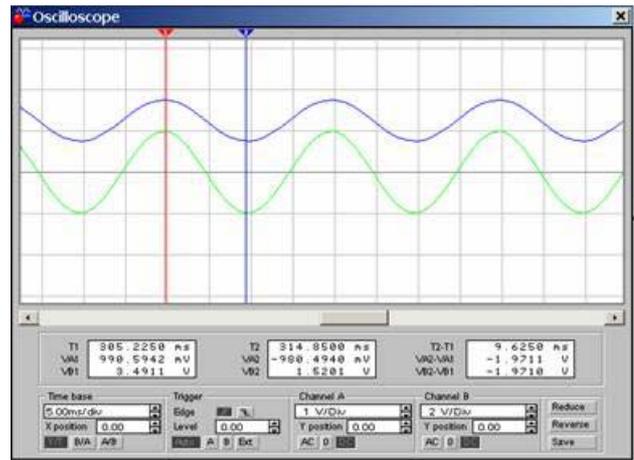


Figure 5.4.1c. Screen dump of oscilloscope (CH A: Input, CH B: Output)

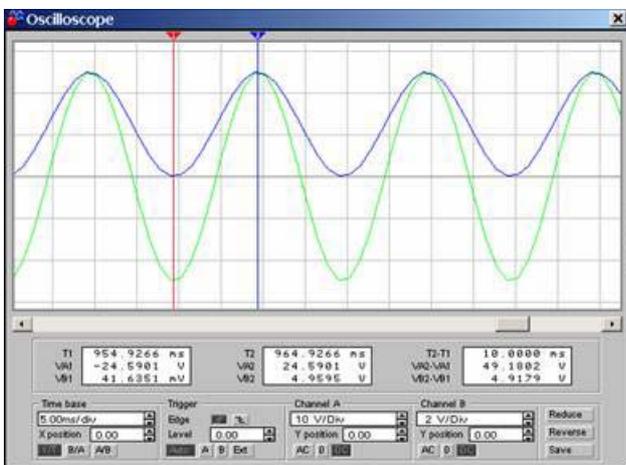


Figure 5.3.1d. Screen dump of oscilloscope (CH A: Input, CH B: Output)

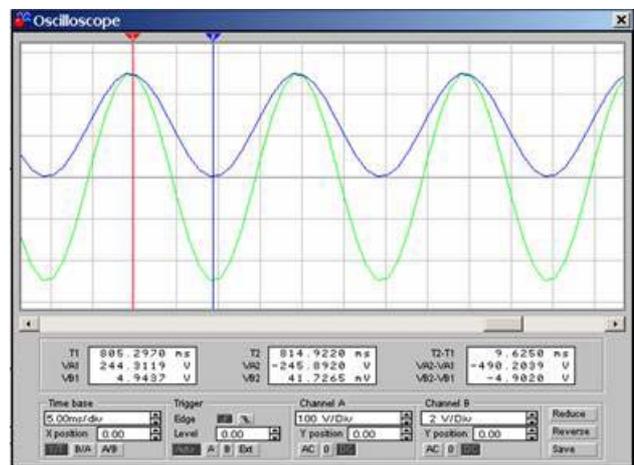


Figure 5.3.1e. Screen dump of oscilloscope (CH A: Input, CH B: Output)

From figure 5.4.1d (using -25 to 25V input) it is clear that a -25 to 25V sine-wave input has been applied and the output is 0 to 5V. Figure 5.4.1e clearly shows a sine-wave input of -250 to 250v (using -250 to 250V input), and an output of to 0V to 5V. Notice the input and output waveforms are the same phase and polarity.

Figure 5.4.1f clearly demonstrates the operation of the diodes, an input of -5 to +5V voltages is applied to the -2.5 to 2.5V input. The peaks of the output waveform have been clipped giving an ADC input voltage range of -0.7 to 5.7V.

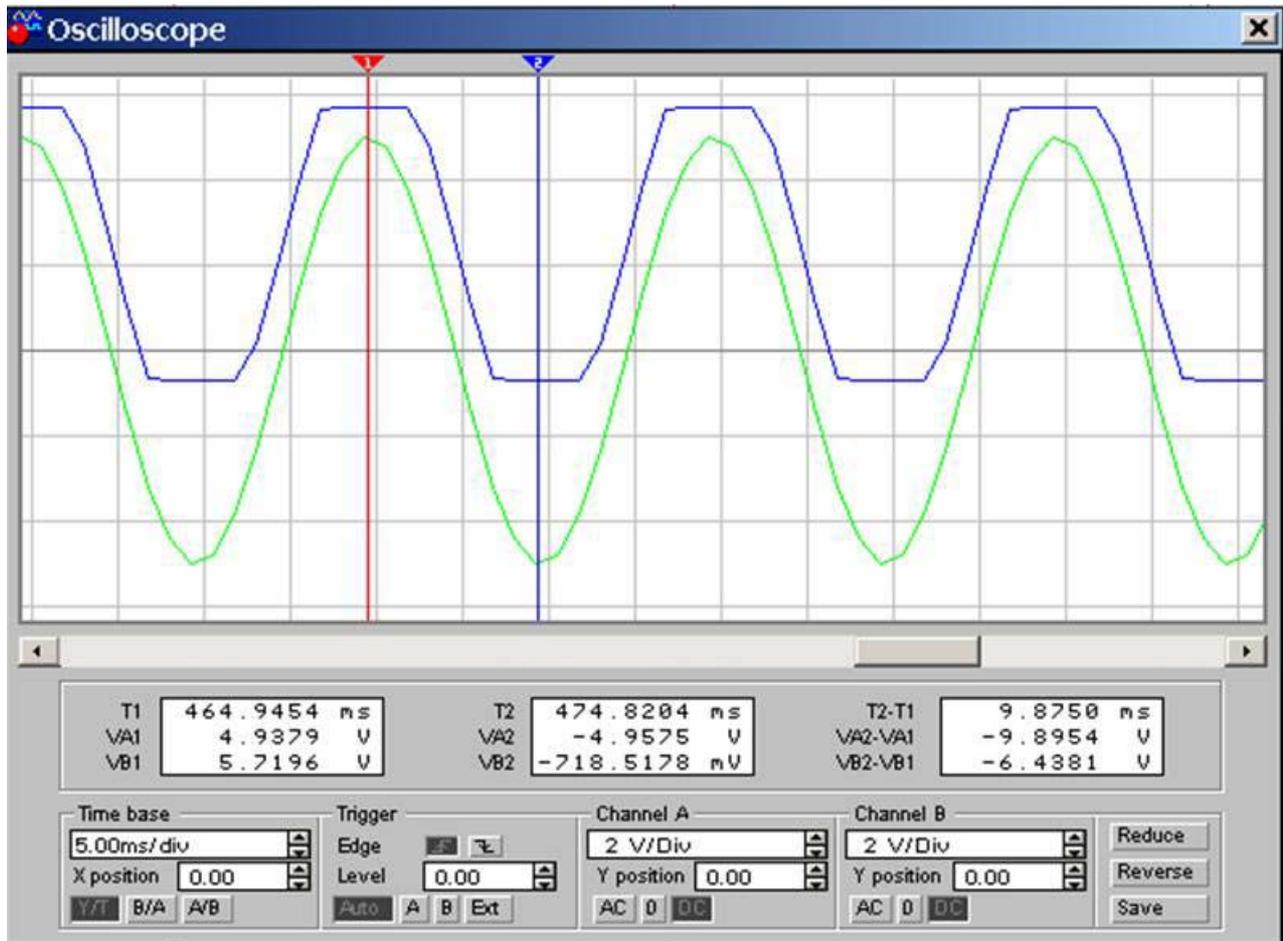


Figure 5.4.1f. Screen dump of oscilloscope (CH A: Input, CH B: Output)

5.5. System Powering Circuit

Figure 5.5a shows the circuit diagram; clearly a 5V regulator is used to generate a 5 volt DC output to power the circuit. The 0.1µF capacitors absorb line noise, while the 100µF capacitors are used for storage in the event of a minor drop in power (milliseconds) the circuit operation will not be affected.

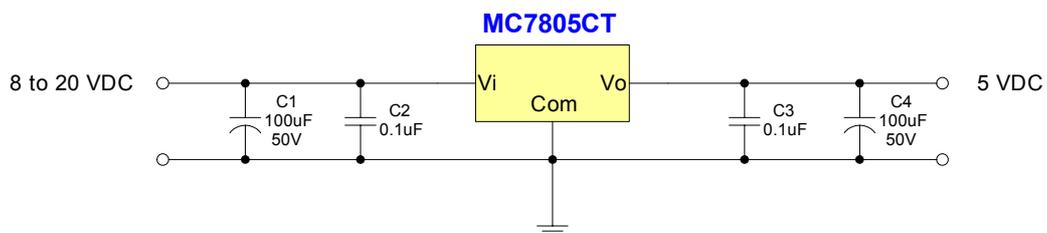


Figure 5.5a. System powering circuit

This means that the circuit has now got a wide operating voltage range as 8 to 20 volts DC will power the circuit. Note there are higher spec 7805 chips available that can operate up to 30 volts DC, if there is a need for a higher voltage range.

The system can be powered from a battery source (e.g. PP3 9V), or a DC power supply (e.g. 12V). It has been decided, not to design a complete power supply unit from scratch, but to use commercially available units. It is important to remember that voltage regulators are not efficient, and as the input voltage increases the least efficient they become, energy is lost in the form of heat. A heat sink is normally required to keep the chip within its maximum operational temperature.

5.6. Cost of Components

All order numbers refer to Farnell catalogue.

Digital Circuit: -

Order Number	Item	Qty.	Cost Each	Cost
325-5573	PIC16F877 – 20/P Microcontroller	1	£7.09	£7.09
407-150	MAX232CPE – RS232 Line Buffer	1	£2.40	£2.40
115-873	HT6264-70 (8k x 8) CMOS SRAM, 70ns acc	1	£1.65	£1.65
170-234	20MHz Crystal A147C	1	£0.75	£0.75
747-014	33pF Ceramic Capacitor	2	£0.13	£0.26
664-315	1µF 25V Electrolytic Capacitor	5	£0.27	£1.35
543-380	1kΩ Metal Oxide Film 1% Resistor	3	£0.02	£0.06
134-3040	40-way dip docket	1	£0.33	£0.33
134-3038	28-way dip socket	1	£0.22	£0.22
134-2988	16-way dip socket	1	£0.12	£0.12
780-091	4-way dip switch	1	£0.40	£0.40
150-812	9-way D-type Socket	1	£0.55	£0.55
			Total	£15.18

Analogue Circuit (Mark 2): -

Order Number	Item	Qty.	Cost Each	Cost
300-373	Quad, 741 op-amp, LM348N	2	£0.40	£0.80
134-2988	16-way dip socket	2	£0.12	£0.12
543-380	1KΩ Metal Oxide Film 1% Resistor	6	£0.02	£0.11
543-627	10KΩ Metal Oxide Film 1% Resistor	9	£0.02	£0.17
543-457	2KΩ Metal Oxide Film 1% Resistor	3	£0.02	£0.06
543-860	100KΩ Metal Oxide Film 1% Resistor	3	£0.02	£0.06
543-860	150Ω Metal Oxide Film 1% Resistor	3	£0.02	£0.06
352-5340	1N4007 Diode	6	£0.04	£0.23
330-760	1W – DC to DC converter 5V input, -15, 15V output	1	£5.05	£5.05
			Total	£6.66

System Powering Circuit: -

Order Number	Item	Qty.	Cost Each	Cost
701-853	MC7805CT, 5V 1A Voltage Regulator	1	£0.30	£0.30
920-757	100µF 50V Electrolytic Capacitor	2	£0.10	£0.21
746-063	100pF Ceramic Capacitor	2	£0.13	£0.26
178-701	Clip On Heat Sink	1	£0.36	£0.36
			Total	£1.13

Total component cost = 15.18 + 6.66 + 1.13 = £22.97

6.0 COMMUNICATION PROTOCOL DEVELOPMENT

There are two main communication protocols: real-time and storage. Note there is an additional communication protocol required to control the PIC (e.g. change sample rate, change from real-time to storage, etc...). Because RS232 has separate transmit and receive lines, there is no chance of a collision with the real-time / storage data. Let's call this additional communication protocol the 'control' protocol, note an interrupt (or PIC must check UART buffer regularly) must be setup in the PIC to let it know that there is a control message waiting to be read.

6.1. Real-Time Mode

Reads one sample at a time sending each reading directly to the PC using RS232, note there is a delay (real-time clock) between readings which is specified by the PC using the 'control' protocol.

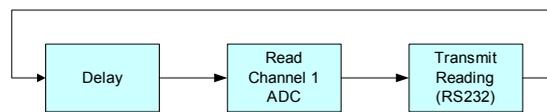


Figure 6.1a. Illustration of real-time mode with one channel

From figure 6.1b it is clear that timing is a big problem, as there are time delays for reading each analogue channel and transmitting the data. This problem is worse at high sample frequencies and one solution is to work around the problem and try to software offset the waveforms to compensate for these time delays. Another option is to use four PICs hence sampling can occur simultaneously, and use a master/slave communication protocol with some way of synchronising all the PICs. PIC microcontrollers are so cheap that this is a real option and was considered, multiple PICs in designs are widely used, unlike microprocessor design where it is usually expected that one microprocessor will control everything (e.g. PC, although there are lots of MPUs in a modern PC besides the Pentium).

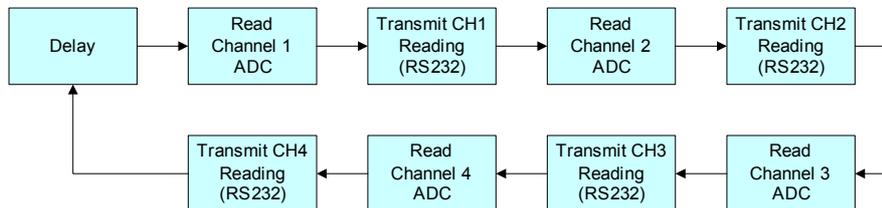


Figure 6.1b. Illustration of real-time mode with four channels (chop mode)

Figure 6.1c illustrates sampling four channels in the real-time mode using alternate sampling. Channel 1 is sampled 1000-times, then channel 2 is sampled 1000-times, then channel 3 is sampled 1000-times, then channel 4 is sampled 1000-times and the process repeats forever.

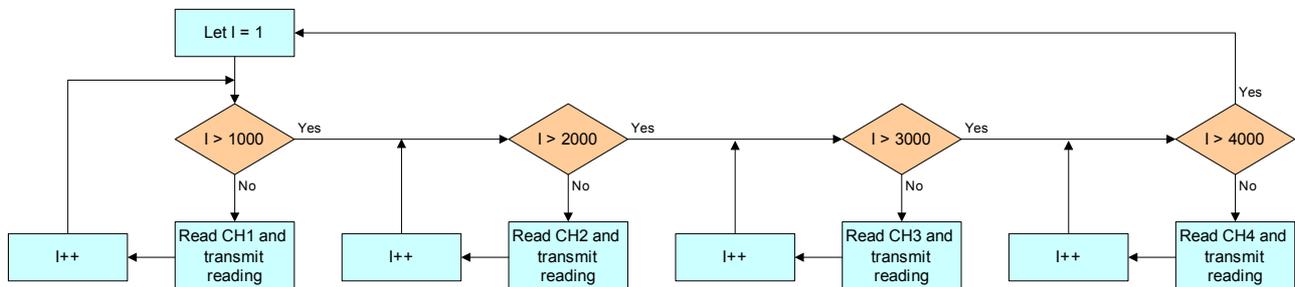


Figure 6.1c. Illustration of real-time mode with four channels (alternate mode)

The advantage of using alternate sampling is that all channels can be sampled at the maximum sample rate, while in chop mode each channel must share the RS232 throughput (RS232 bottleneck), for example say the maximum sample rate was 4KHz (set by the maximum RS232 frame structure throughput), one channel enabled would have a max sample rate of 4KHz, two channels enabled would have a max sample rate of 2KHz each, four channels enabled would have a max sample rate of 1KHz each.

But alternate has a disadvantage; timing information between different channels (e.g. channel 1 leads channel 2 by 25ms) is lost. It is possible to software correct this problem, for example using one of the PICs real-time timers it is known how much time as elapsed between the sampling of channels, this information could be transmitted to the PC Scope program which then draws waveforms with the correct time gap between channels. Another disadvantage is that continuous monitoring is impossible as there are gaps in channel data (e.g. continuous monitoring required for an ECG signal). It is clear that alternate mode is only useful for periodical waveforms (e.g. sinewave), else chop mode should be used.

6.2. Storage Mode

Read a finite number of samples, storing them into RAM (external RAM chip). The readings are then transferred to the PC in one large block, hence removing the RS232 bottleneck and providing faster sample rates.

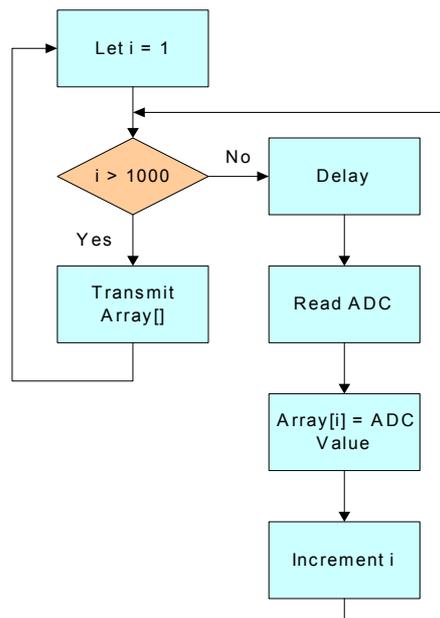


Figure 6.2a. Illustration of storage mode with one channel

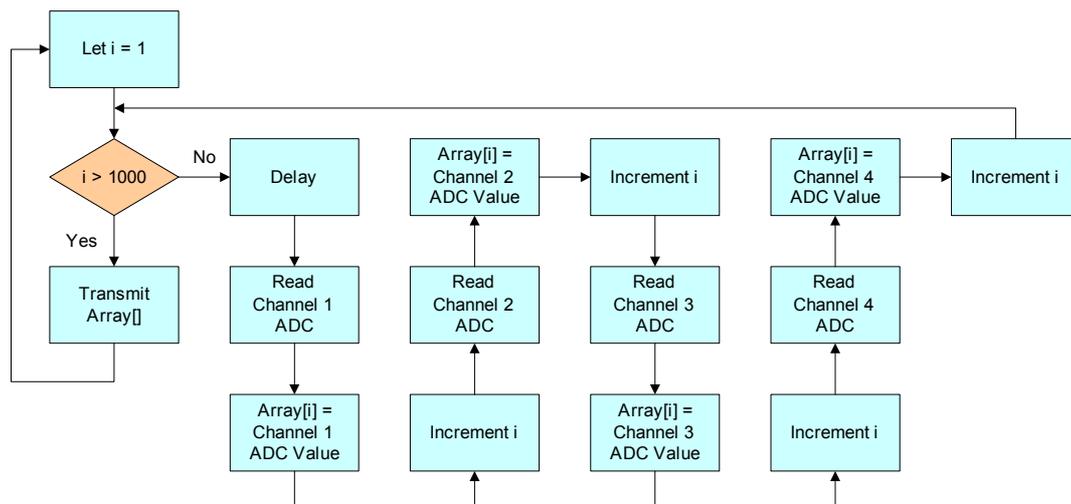


Figure 6.2b. Illustration of storage mode with four channels (chop mode)

Alternate sampling can applied to storage mode, but there is no real advantage over chop mode. Recall that the main reason of having alternate mode was to maximise the sample rate of each channel in the real-time mode. This no longer apples as storage mode removes the RS232 bottleneck, the new bottleneck is now the speed of the PICs ADC, hence alternate mode cannot increase channel sample rates.

6.3. Control Protocol

The PIC checks the UART buffer regularly (or UART interrupt) checking to see if a control message is being received. Once a control message has been detected, the PIC leaves the main program and checks (CRC check) that the message is valid, if not return to the main program, if valid decode and implement the message. This process is illustrated in figure 6.3a.

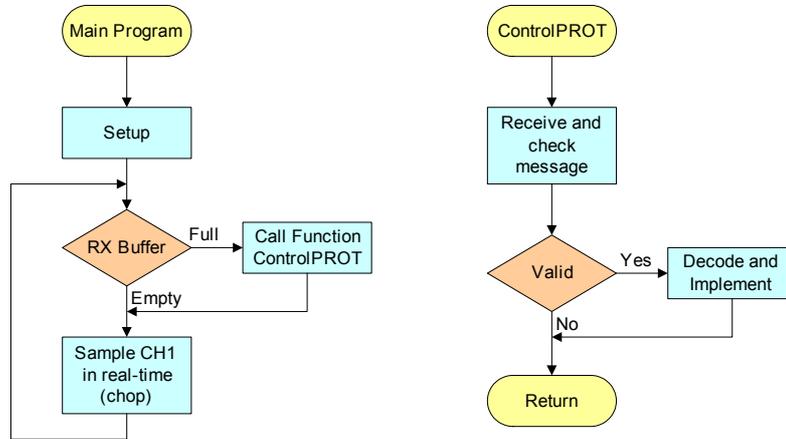


Figure 6.3a. Illustration of the control protocol

6.4. Frame Structure: Real-Time Mode

10-bit ADC, hence 2-bytes are sent per reading.

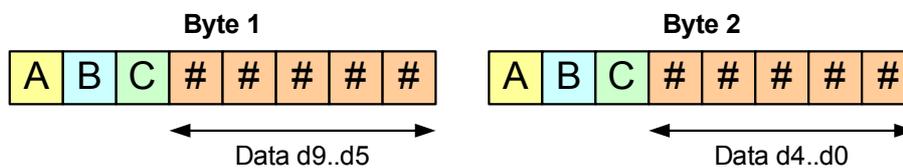


Figure 6.4a. Frame structure: Real Time.

If A = 0, byte 1 which contains data d9...d5 (Upper 5-bits of ADC reading).
 If A = 1, byte 2 which contains data d4...d0 (Lower 5-bits of ADC reading).

B & C specify what channel the data is for.

B	C	
0	0	Channel 1
0	1	Channel 2
1	0	Channel 3
1	1	Channel 4

Since the RS232 communications is the bottleneck, it is important that the real-time communication protocol is as efficient as possible. At first it appears that the structure is 100% efficient as all bytes are used, and each one extremely important.

Yes this is partly true; however the only data of real value is the ADC readings, the reset are for identification purposes only. Hence the real efficiency of this frame structure is (10 data bits, 16 bits in total, hence efficiency = 10/16 x 100) 62.5%.

It is theoretically possible to use a frame structure that is 100% efficient, but this requires (because there is no identification code) extremely good synchronisation between the PC and the PIC. The main problem with this is that if a byte is lost (due to noise, bad line, buffer overflow, etc...) it would be extremely difficult to recover. The other option is to have the PC request each and every reading (Master/Slave), the problem with this is that there is a delay sending the request to the PIC and a processing delay hence this method would be much slower in reality.

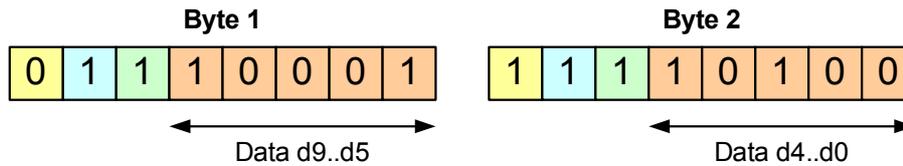


Figure 6.4b. Real-time frame structure example, channel 4 reads 564d (10,0011,0100b) from ADC

Notice there is no check sum or CRC, the reason for this is to maximise the data throughput. Clearly there is a problem; however the frame structure is designed in such a way that some errors can be detected. Note the cable between the MAX232 buffer (connect to PIC) and the PC will be high quality cable (short length) and errors should not occur that frequently anyway.

A few rules that must occur for the frame to be valid (non-valid frames are flushed): -

1. Byte 1 is always followed by byte 2 for a particular channel.
 E.g. B1_CH1 ← B2_CH1 ← B1_CH3 ← B2_CH3 ← OK
 E.g. B1_CH1 ← B1_CH3 ← B2_CH3 ← ... Error (no B2_CH2) flush this byte
2. Channel readings can occur in any order as long as byte 2 follows byte 1.
 E.g. B1_CH3 ← B2_CH3 ← B1_CH2 ← B2_CH2 ← ...

If one or two readings are lost, it should not affect the appearance of the waveform too much, as there are at least 10 readings per cycle and probably at lot more at high sampling rates. Also because it is real-time the screen is continuously refreshed, hence the error will probably not stay on the screen for very long (a couple of milliseconds) until it is replaced by new readings.

6.5 Sample Rate: Real-Time Mode

The RS232 communications is the bottleneck as the maximum sampling frequency of PIC16F877 microcontroller ADC is higher than can be transmitted over RS232. Therefore it makes since to calculate the maximum sample rate based on how much data can be transmitted through RS232 at different baud rates.

Two bytes are sent per frame, note each byte contains 8-bits plus a start and stop bit. Hence 20-bits are sent per reading.

$$\text{Sample Rate} = \frac{\text{Baud Rate}}{20 \times \text{No .of Channels}}$$

Number of Channels	Max Sample Rate at 115 Kbps	Max Sample rate at 56.6 Kbps	Max Sample Rate at 33.6 Kbps
1	5.75 KHz	2.83 KHz	1.68 KHz
2	2.87 KHz	1.41 KHz	840 Hz
3	1.91 KHz	0.94 KHz	560 Hz
4	1.43 KHz	0.70 KHz	420 Hz

Figure 6.5a. Table showing the theoretical real-time sample rates (chop, 10-bit ADC).

Note that the values shown in Figure 6.5a assume that there is no time delay between readings. Clearly this is not the case in practice and the actual maximum sample rates will be lower than specified e.g. PIC takes 19.72µS (see datasheet for PIC16F877) to acquire each sample, plus there will be processing delays. However these values should be reasonably accurate because the hardware UART is being used, hence the PIC can sample the next reading while the UART is transmitting the last reading.

Note if alternating sampling (not chop) is used the sample rate will not decrease with the number of channels, for example figure 6.5a states that the maximum sample rate for one channel at 115 kbps is 5.75 KHz, if alternate mode is used this maximum sampling rate still applies even if 4 channels are enabled. Recall that alternate mode is only useful for periodical waveforms.

The PIC16F877 has an 10-bit ADC, if speed is important the resolution could be reduced to 8-bits when sampling four channels. For example channels 1,2,3,4 are always transmitted along with sync frame (5-bytes for 4 channels, 80% efficient), hence the following sample rates are now possible: -

$$\text{Sample Rate (8-bit ADC, Four Channels)} = \frac{\text{Baud Rate}}{50}$$

Number of Channels	Max Sample Rate at 115 Kbps	Max Sample rate at 56.6 Kbps	Max Sample Rate at 33.6 Kbps
1	2.3 KHz	1.132KHz	672 Hz
2	2.3 KHz	1.132KHz	672 Hz
3	2.3 KHz	1.132KHz	672 Hz
4	2.3 KHz	1.132KHz	672 Hz

Figure 6.5b. Table showing the theoretical real-time sample rates (chop, 8-bit ADC).

Another possible method for increasing real-time sampling rates is to use two or more serial links. For example most PCs have two RS232 ports (more can be added by inserting serial cards), why not use both serial ports to receive the data. Obviously the PIC only has one hardware UART, hence the solution is to use the hardware UART plus a software UART (CSS C compiler automatically generates the code when non-hardware PINs are selected, which is reconfigurable throughout the program). Two bytes are sent (10-bit ADC) for each channel, hence put the first byte on COM1 and the second on COM2, this will double the maximum sampling rate. Note the MAX232CPE chip is a dual RS232 line driver chip hence no extra hardware is required, allow the additional software UART will push the PIC to the limits (should be OK at 20MHz).

$$\text{Sample Rate (Multipliable Serial Links)} = \frac{N_{\text{Serial Links}} \times \text{Baud Rate}}{\text{No. of Channels} \times 20}$$

Number of Channels	Max Sample Rate at 230 Kbps (2 serial links at 115 Kbps)	Max Sample rate at 115 Kbps (2 serial links at 56.6 Kbps)	Max Sample Rate at 67.2 Kbps (2 serial links at 33.6 Kbps)
1	11.5 KHz	5.75 KHz	3.36 KHz
2	5.75 KHz	2.87 KHz	1.68 KHz
3	3.83 KHz	1.91 KHz	1.12 KHz
4	2.88 KHz	1.43 KHz	840 Hz

Figure 6.5c. Table showing the theoretical real-time sample rates (chop, 10-bit ADC, dual RS232 cable).

For faster sample rates, storage mode must be used, for example the PIC samples a waveform at 20 KHz storing each reading into an array (e.g. external RAM chip). Then once a certain number of samples have been taken (say 200) the PIC stops sampling and sends the readings to the PC (via RS232) in one large block (including CRC or check sum), once ACK (Acknowledgement) is received from the PC the PIC starts sampling again and the process repeats forever. Another option is to use a different transport medium (e.g. USB / Parallel) which would allow for faster real-time sample rates.

Note the real-time communication protocol does not rely on the transport medium, hence if the transport medium is changed the real-time protocol should be OK. Expect for maybe USB as USB ports can be shared with other devices (many devices connected to the same port e.g. printer, scanner, mouse, etc...) hence the communication protocol must be compliant with the USB standard.

6.6. Frame Structure: Storage Mode

Figure 6.6a shows the frame structure for the storage mode, each field is 1 byte in size, expect for the 'data' field which is variable and its size is specified in the 'count' field.

Since each frame could contain up to 255 data bits (large block of data) it is much more likely that an error will error when compared to the short real-time frames. Hence the need for error detection, at present a 16-bit CRC is included in the frame which will detect almost every possible error (including burst errors).

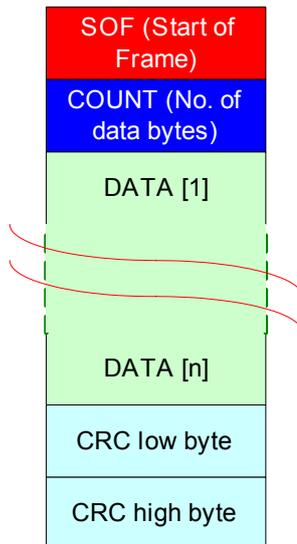


Figure 6.6a. Frame Structure: Storage Mode

But the generation of the CRC is processor hungry; at first it was thought that it may not be suitable with the PIC. Although it is possible to generate the CRC using a lookup table, this reduces processor load, but this lookup table uses a lot of RAM.

The plan is to use a 16-bit CRC for error detection, as this practically guarantees that all errors are detected. But if it turns out that it is impractical to implement on the PIC, a simple checksum will be used instead, but the disadvantage of this is that not all errors will be detected (burst errors will be a problem). Note all fields in the frame are included in the CRC (or maybe checksum) calculation e.g. SOF, COUNT, and DATA[i] ... DATA[n].

The 'data' field uses the same frames as specified for the real-time frame structure. The big advantages of this is that the code reusable e.g. subroutines/functions can be written and used in the real-time and storage modes.

There are two options for checking if the frame is correct: -

1. Calculated the CRC for the received frame and compare with the received CRC.
2. Send the entire frame to the CRC function (including received CRC) and if the calculated CRC is zero then the frame is correct.

6.7. Sample Rate: Storage Mode

Since operating in storage mode, the maximum sampling rate is only limited by the maximum sampling rate of the PIC ADCs and not the RS232 link. The datasheet for the PIC16F877 specifies acquisition time (TACQ): -

$$\begin{aligned} TACQ &= \text{Amplifier Setting Time} + \text{Hold Capacitor Charging Time} + \text{Temperature Coefficient} \\ &= T_{AMP} + T_C + T_{COFF} \\ &= 2\mu\text{S} + T_C + [(\text{Temperature} - 25^\circ\text{C})(0.05\mu\text{S}/^\circ\text{C})] \end{aligned}$$

$$\begin{aligned} T_C &= C_{HOLD} (R_{IC} + R_{SS} + R_S) \ln(1/2047) \\ &= -120\text{pF}(1\text{k}\Omega + 7\text{k}\Omega + 10\text{k}\Omega) \ln(0.0004885) \\ &= 16.47\mu\text{S} \end{aligned}$$

$$\begin{aligned} TACQ &= 2\mu\text{S} + 16.47\mu\text{S} + [(50^\circ\text{C} - 25^\circ\text{C})(0.05\mu\text{S}/^\circ\text{C})] \\ &= 19.72\mu\text{S} \end{aligned}$$

Therefore theoretically the PICs ADC can sample at $(1/19.72\mu\text{S})$ 50.7 KHz. But according to the datasheet "after a conversion has completed, a $2.0 T_{AD}$ delay must be complete before acquisition can begin again. During this time, the holding capacitor is not connected to the selected A/D input channel." [D1]

T_{AD} is defined as the A/D conversion time per bit. For correct A/D conversions, the A/D conversion clock (T_{AD}) must be selected to ensure a minimum T_{AD} time of $1.6\mu\text{s}$. Assuming a minimum A/D conversion clock (T_{AD}) time of $1.6\mu\text{s}$ (e.g. the PIC is running at 20 MHz with 32TOSC selected for T_{AD}), the maximum sample rate is: -

$$S_{MAX} = \frac{1}{TACQ + 2T_{AD}} = \frac{1}{22.92\mu\text{s}} = 43.63\text{KHz (Ksps)}$$

For faster sample rates an external ADC with direct memory access is required.

6.8. CRC16 (Cyclical Redundancy Check) used in Storage Mode

Procedure: -

1. Load a 16-bit register with FFFF hex (all '1's). Call this the CRC register.
2. Exclusive OR the first 8-bit bytes of the message with the low order byte of the 16-bit CRC register, putting the result in the CRC register.
3. Shift the CRC register one bit to the right, zero-filling the MSB. Extract and examine the LSB.
4. If the LSB was 0: Do nothing – proceed to step 5.
If the LSB was 1: EXOR the CRC register with the polynomial value 0xA001.
5. Repeat steps 3 and 4 until 8 shifts have been performed. When this is done, a complete 8-bit byte will have been processed.
6. Repeat steps 2 through 5 for the next 8-bit byte of the message. Continue doing this until all the bytes have been processed.
7. The final content of the CRC register is the CRC value.
8. Swap low & high bytes of CRC, Return CRC.

C code for producing CRC (pure calculation, no lookup table, PC Code): -

```

unsigned short CRC16(unsigned char *message, unsigned short data_length)
{
    unsigned short CRC = 0xFFFF,           // CRC initialised with all '1's
                   i,                       // Loop Control
                   temp;                    // Temp for swap

    while (data_length--)                  // Loop until all bytes calculated
    {
        CRC=CRC^(*message++);

        for (i=0;i++<8;)                 // Shift all 8-bits
        {
            if (CRC & 0x0001) CRC = (CRC>>1) ^0xA001;
            else CRC>>=1;
        }

        temp = CRC & 0xFF;                // Temp storage for swap
        CRC = ((CRC>>8)&0xFF) + (temp<<8); // Swap high and low byte
    }

    return (CRC);                          // Return calculated 16-bit CRC
}

```

C code for producing CRC (using lookup table): -

```

unsigned short CRC16(unsigned char *message, unsigned short data_length)
{
    /* Table of CRC values for high-order byte */
    static unsigned char Table_CRC_Hi[] = {
        0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC1, 0X81, 0X40, 0XC1, 0X81, 0X40, 0X01, 0XC0,
        0X40, 0X01, 0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41,
        0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC1, 0X81, 0X40, 0X00, 0XC1, 0X81, 0X40, 0X01,
        0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41,
        0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0,
        0X40, 0X01, 0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0,
        0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01,
        0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41,
        0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0,
        0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0,
        0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X01,
    }
}

```

```

0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81, 0X40, 0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41,
0X00, 0XC1, 0X81, 0X40, 0X01, 0XC0, 0X80, 0X41, 0X01, 0XC0, 0X80, 0X41, 0X00, 0XC1, 0X81,
0X40};

/* Table of CRC values for low-order byte */
static char Table_CRC_Lo[] = {
0X00, 0XC0, 0XC1, 0X01, 0XC3, 0X03, 0XC2, 0XC6, 0X06, 0X07, 0XC7, 0X05, 0XC5, 0XC4,
0X04, 0XCC, 0X0C, 0X0D, 0XCD, 0X0F, 0XCF, 0XCE, 0X0E, 0X0A, 0XCA, 0XCB, 0X0B, 0XC9, 0X09,
0X08, 0XC8, 0XD8, 0X18, 0X19, 0XD9, 0X1B, 0XDB, 0XDA, 0X1A, 0X1E, 0XDE, 0XDF, 0X1F, 0XDD,
0X1D, 0X1C, 0XDC, 0X14, 0XD4, 0XD5, 0X15, 0XD7, 0X17, 0X16, 0XD6, 0XD2, 0X12, 0X13, 0XD3,
0X11, 0XD1, 0XD0, 0X10, 0XF0, 0X30, 0X31, 0XF1, 0X33, 0XF3, 0XF2, 0X32, 0X36, 0XF6, 0XF7,
0X37, 0XF5, 0X35, 0X34, 0XF4, 0X3C, 0XFC, 0XFD, 0X3D, 0XFF, 0X3F, 0X3E, 0XFE, 0XFA, 0X3A,
0X3B, 0XFB, 0X39, 0XF9, 0XF8, 0X38, 0X28, 0XE8, 0XE9, 0X29, 0XEB, 0X2B, 0X2A, 0XEA, 0XEE,
0X2E, 0XEF, 0XEF, 0XED, 0XED, 0XEC, 0X2C, 0XE4, 0X24, 0X25, 0XE5, 0X27, 0XE7, 0XE6, 0X26,
0X22, 0XE2, 0XE3, 0XE1, 0X21, 0X20, 0XE0, 0XA0, 0X60, 0X61, 0XA1, 0X63, 0XA3, 0XA2,
0X62, 0X66, 0XA6, 0XA7, 0X67, 0XA5, 0X65, 0X64, 0XA4, 0X6C, 0XAC, 0XAD, 0X6D, 0XAF, 0X6F,
0X6E, 0XAE, 0XAA, 0X6A, 0X6B, 0XAB, 0X69, 0XA9, 0XA8, 0X68, 0X78, 0XB8, 0XB9, 0X79, 0XBB,
0X7B, 0X7A, 0XBA, 0XBE, 0X7E, 0X7F, 0XBF, 0X7D, 0XBD, 0XBC, 0X7C, 0XB4, 0X74, 0X75, 0XB5,
0X77, 0XB7, 0XB6, 0X76, 0X72, 0XB2, 0XB3, 0X73, 0XB1, 0X71, 0X70, 0XB0, 0X50, 0X90, 0X91,
0X51, 0X93, 0X53, 0X52, 0X92, 0X96, 0X56, 0X57, 0X97, 0X55, 0X95, 0X94, 0X54, 0X9C, 0X5C,
0X5D, 0X9D, 0X5F, 0X9F, 0X9E, 0X5E, 0X5A, 0X9A, 0X9B, 0X5B, 0X99, 0X59, 0X58, 0X98, 0X88,
0X48, 0X49, 0X89, 0X4B, 0X8B, 0X8A, 0X4A, 0X4E, 0X8E, 0X8F, 0X4F, 0X8D, 0X4D, 0X4C, 0X8C,
0X44, 0X84, 0X85, 0X45, 0X87, 0X47, 0X46, 0X86, 0X82, 0X42, 0X43, 0X83, 0X41, 0X81, 0X80,
0X40};

unsigned char CRC_Hi = 0xFF;          /* High byte of CRC initialised */
unsigned char CRC_Lo = 0xFF;          /* Low byte of CRC initialised */
unsigned index=0;                     /* Will index into CRC lookup table */

while (data_length--)
{
    index = CRC_Hi ^ *message++;      /* Calculate the CRC */
    CRC_Hi = CRC_Lo ^ Table_CRC_Hi[index];
    CRC_Lo = Table_CRC_Lo [index];
}

return (CRC_Hi << 8 | CRC_Lo);
}

```

Example CRC calculation: -

```

Message Length:      1
Message [0]:         0000 0010

```

Initialised CRC with all '1's

```
CRC = 1111 1111 1111 1111
```

Exclusive OR the first 8-bytes of the message with the low order byte of the 16-bit CRC register, putting result into the CRC.

```

CRC:      1111 1111 1111 1111
Message:  0000 0000 0000 0010
NEW CRC:  1111 1111 1111 1101

```

LSB is '1', >>1 (no. 1) & EOR with A001.

```

>>1:     0111 1111 1111 1110
A001:    1010 0000 0000 0001
NEW CRC: 1101 1111 1111 1111

```

LSB is '1', >>1 (no. 2) & EOR with A001.

```

>>1:     0110 1111 1111 1111
A001:    1010 0000 0000 0001
NEW CRC: 1100 1111 1111 1110

```

LSB is '0', >>1 (no. 3).

```
>>1:     0110 0111 1111 1111
```

LSB is '1', >>1 (no. 4) & EOR with A001.

```

>>1:     0011 0011 1111 1111
A001:    1010 0000 0000 0001
NEW CRC: 1001 0011 1111 1110

```

LSB is '0', >>1 (no. 5).

```
>>1:     0100 1001 1111 1111
```

LSB is '1', >>1 (no. 6) & EOR with A001.

```

>>1:     0010 0100 1111 1111
A001:    1010 0000 0000 0001

```

```

NEW CRC:      1000 0100 1111 1110

LSB is '0', >>1 (no. 7).
>>1:        0100 0010 0111 1111

LSB is '1', >>1 (no. 8) & EOR with A001.
>>1:        0010 0001 0011 1111
A001:       1010 0000 0000 0001
NEW CRC:    1000 0001 0011 1110   = 0x813E

Swap high & Low Bytes.
Cal CRC:    16001 decimal or 3E81 Hex

```

6.8.1 Comparison of Raw Calculation CRC & Lookup Table CRC

Theoretically the raw calculation CRC function produces its result slower than the lookup table CRC function, but the lookup table will eat-up program memory (512 words). The objective of this simple experiment is to prove that this is the case and to find out exactly the performance of each, this information will be used to help decide, on which technique is best for this project. Simulations were carried out using Microchip MPLAB.

Raw calculation CRC test program (crc.c): -

```

#include <16F877.h>

long int CRC16(char *message, int length);

main()
{
    long int crcANS;
    char test[5];

    test[0] = 5;
    test[1] = 4;
    test[2] = 3;
    test[3] = 2;
    test[4] = 1;

    crcANS = CRC16(test,5);

    Sleep();
}

long int CRC16(char *message,int length)
{
    /* 16-bit var's */
    long int CRC = 0xFFFF,          /* CRC initialised with all '1's */
              i,                    /* Loop Control */
              temp;                 /* Temp for swap */

    while (length--)                /* Loop until all bytes calculated*/
    {
        CRC=CRC^(*message++);

        for (i=0;i++<8;)            /* Shift all 8-bits */
        {
            if (CRC & 0x0001) CRC = (CRC>>1) ^0xA001;
            else CRC>>=1;
        }

        temp = CRC & 0xFF;           /* Temp storage for swap */
        CRC = ((CRC>>8)&0xFF) + (temp<<8); /* Swap high and low byte */
    }

    return (CRC);                   /* Return calculated 16-bit CRC */
}

```

Notice data types have been changed from that used in the PC version, e.g. char is 8-bit, short is 16-bit, int is 32-bit and long is 64-bit on a PC, while char is 8-bit, short is 1-bit, int is 8-bit, and long is 16-bit on a PIC.

Abstract from crc.lst: -

```

MPASM

CCS PCM C Compiler, Version 2.707, 8851

        Filename: C:\WORK\COLIN\MPLAB\PROJECT\CRC\CRC.LST

        ROM used: 116 (1%)
                Largest free fragment is 2048
        RAM used: 12 (7%) at main() level
                22 (13%) worst case
        Stack:   1 locations

        ...
        ...

```

Test array setup containing 5 elements, the test array is then send to the CRC function which then calculates the 16-bit CRC for the test array. Break point was inserted at the position of the Sleep () command, the stopwatch windows was opened and reset. Program was then simulated (stopwatch starts) until the break point is reached (stopwatch stops). Note the stopwatch time is the total time taken to execute all instructions in the test program and not just the CRC16 function. This is fine as the two test programs are identical except for the CRC16 function, therefore a good comparison can be made.

The screen dump of MPLAB after simulation (Raw Calculation): -

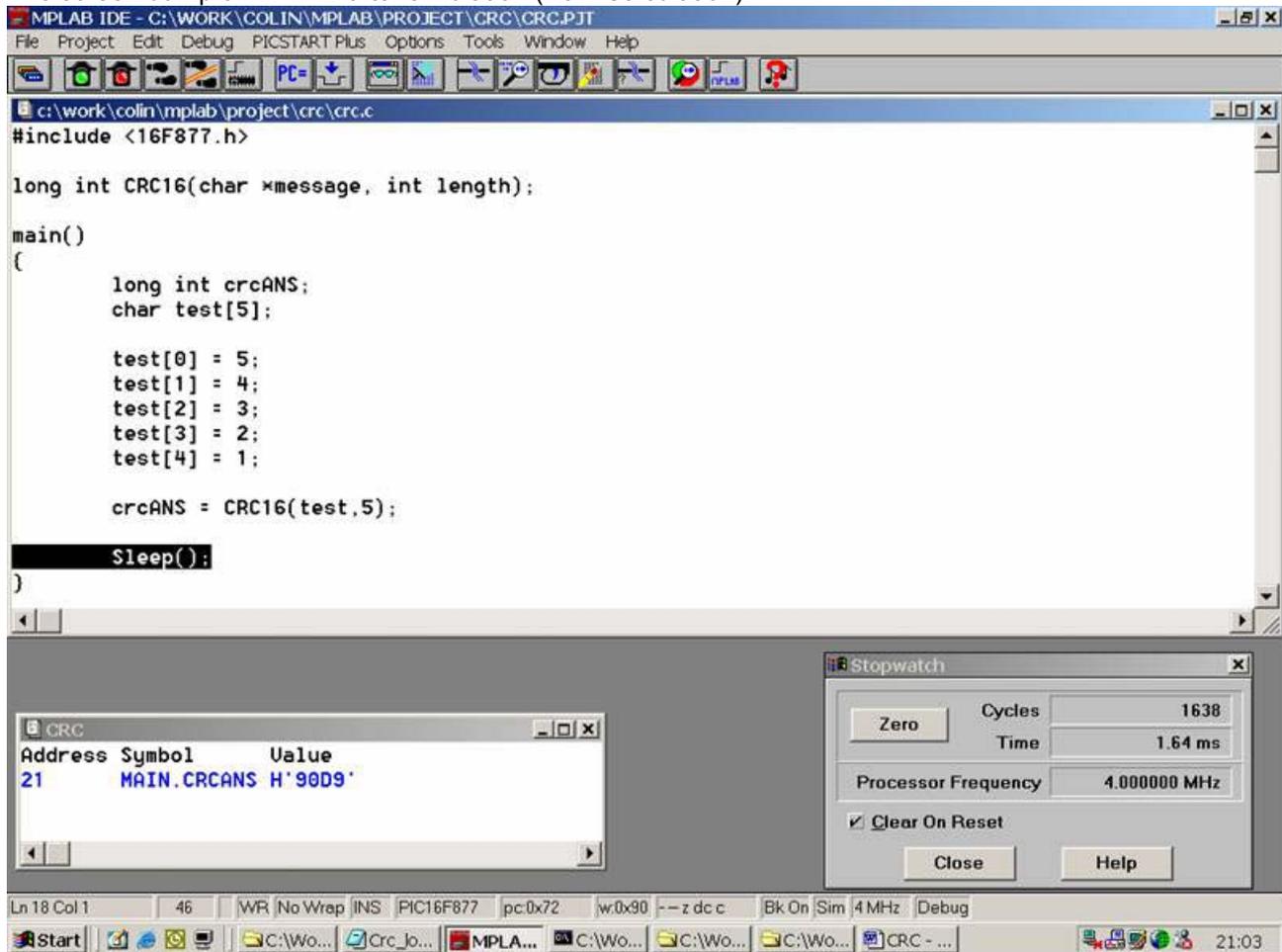


Figure 6.8.1a. Result of raw calculation CRC test

Total execution time for the raw calculation CRC test program was 1.64ms @4Mhz that's 1638 Cycles.

Lookup table CRC test program (crcLT.c): -

```

#include <16F877.h>

/* Table of CRC values for high-order byte */
const char Table_CRC_Hi[256] = {

```


Abstract from crclT.lst: -

MPASM

CCS PCM C Compiler, Version 2.707, 8851

Filename: C:\WORK\COLIN\MPLAB\PROJECT\CRC\CRCLT.LST

```

ROM used: 600 (7%)
          Largest free fragment is 2048
RAM used: 12 (7%) at main() level
          21 (12%) worst case
Stack:    2 locations

```

```

c:\work\colin\mplab\project\crc\crclt.c
test[1] = 4;
test[2] = 3;
test[3] = 2;
test[4] = 1;

crcANS = CRC16(test,5);

Sleep();
}

long int CRC16(char *message,int length)
{
    char CRC_Hi = 0xFF; /* High byte of CRC initialised */
    char CRC_Lo = 0xFF; /* Low byte of CRC initialised */
    long int CRC;
    int index=0; /* Will index into CRC lookup table */

```

Address	Symbol	Value
21	MAIN.CRCANS	H'90D9'

Stopwatch

Zero	Cycles	286
	Time	286.00 us
Processor Frequency		4.000000 MHz
<input checked="" type="checkbox"/> Clear On Reset		
Close		Help

Figure 6.8.1b. Result of lookup table CRC test

Total execution time for the lookup table CRC test program was 286.00 μ s @4Mhz that's 286 Cycles.

Clearly the lookup table CRC (286 μ s) runs over 5 times (5.7) faster than the raw calculation CRC (1.64ms) for this example. But unfortunately it is also clear that the lookup table CRC used 600 words of program memory (7% of total) and the raw calculation CRC only used 116 words (1% of total) hence the CRC lookup table uses 6 times more program memory than the raw calculation version. Obviously the reason for this is because the lookup table CRC uses 512 words of program memory for its lookup table.

Evidently if speed is important and there is plenty of program memory available, the lookup table CRC is the one to use. But if program memory is at a premium and speed is not important the raw calculation CRC is the one to use.

6.9. Frame Structure: Control Protocol

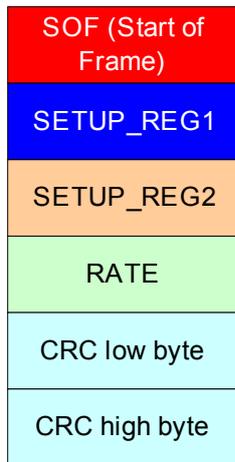


Figure 6.9a. Frame structure: control protocol

Figure 6.9a shows the frame structure for the control protocol, each field is 1 byte in size. Since this control protocol is used to directly control the PIC, it is important that the PIC is able to check the frame before processing the information (e.g. the PIC should ignore random noise or corrupt frames). Hence a 16-bit CRC (same as used for storage mode) is used to make sure that the frame is valid before the PIC implements the control message.

The CRC takes some time to calculate hence there are crude checks that are carried out first before a CRC is calculated, as false control messages (random noise) would severely affect system performance. Hence the PIC waits for the SOF frame and ignores all other characters until a frame is detected, once a frame is detected it then waits for 5 more bytes. If these 5 bytes are not received within a certain time (e.g. max of 1.5ms gap between bytes) the PIC will timeout and return to what it was doing before the message was detected. If 5 bytes are received the frame is CRC check and if valid the message is decoded and implemented. Note it is possible that random noise could generate a SOF (e.g. 0x01) and 5 further bytes, but the CRC will make sure that the false frame is not implemented.

b7	b6	b5	b4	b3	b2	b1	b0
M	R1	R0	I	C4	C3	C2	C1
SETUP_REG1							
M = 0: Real-time Mode M = 1: Storage Mode							
R1	R0	RATE Multiplier					
0	0	X1, e.g. 1 to 255 Hz					
0	1	X10, e.g. 10 to 2550 Hz					
1	0	X100, e.g. 100 to 25,500 Hz					
1	1	X1000, e.g. 1 kHz to 255 kHz					
I = 0: Sample when PIC is awake I = 1: Sample when PIC is sleeping							
Cn = 0: Channel n disabled Cn = 1: Channel n enabled							

Figure 6.9b. SETUP_REG1

SETUP_REG1 is shown in figure 6.9b, this register specifies real-time/storage mode, sample rate multiplier, sample awake/sleeping and which channels are enabled / disabled. Note sampling when the PIC is sleeping is only an option for very slow sample rates as it takes a long time for the PIC to return to full speed after waking up. Recall that sampling when the PIC is sleeping results in more accurate ADC readings (less noise)

SETUP_REG2 is shown in figure 6.9c, this register specifies the number of samples to be stored into RAM (storage mode only), chop or alternate mode and the voltage input range (change gain of op-amps). Notice that b7 is marked as 'R', this means that the bit is not used and is reserved for future use.

b7	b6	b5	b4	b3	b2	b1	b0
R	V3	V2	V1	S	S2	S1	S0
SETUP_REG2							
				S2	S1	S0	Number of Samples
				0	0	0	100
				0	0	1	250
				0	1	0	500
				0	1	1	750
				1	0	0	1000
				1	0	1	2000
				1	1	0	3000
				1	1	1	4000
<i>* Storage mode only, no effect in real-time mode.</i>							
S = 0: Chop Mode S = 1: Alternate Mode							
V2		V1		V0		Voltage Input Range	
0		0		0		-1 to +1V	
0		0		1		-2.5 to + 2.5V	
0		1		0		-5 to +5V	
0		1		1		-10 to +10V	
1		0		0		-25 to +25V	
1		0		1		-50 to +50V	
1		1		0		-100 to +100V	
1		1		1		-250 to +250V	
R: Reserved for future use							

Figure 6.9c. SETUP_REG2

Field RATE in figure 6.9a specifies the required sample rate, this figure is multiplied by the RATE multiplier that is specified in SETIP_REG1 (figure 6.9b). For example if RATE is 100 and RATE multiplier is 10 (R1 = 0, R0 = 1) the specified sample rate is 100 x 10 = 1000 Hz.

7.0. THE SCOPE PROGRAM

This Windows based program, graphically displays waveforms without flicker, it has a good user interface that is easy to use, and directly communicates with the PIC. RS232 transport medium has been selected, this medium is easy to program, reliable and every PC comes with at least one RS232 port. But it is slow and will limit the maximum sampling rate in the real-time mode. Therefore it is important that the application is design to be flexible, as its probable that the program will be modified some time in the future for use with another medium (e.g. Parallel, USB, etc...).

The development of this program is not fully completed, but has progressed sufficiently to display real-time waveforms using various triggering methods, selectable time-base, selectable voltage scales, etc... Storage mode operation and direct control of the PIC has not been implemented yet, as well as some advantaged features like: streaming real-time data to disk, automatic frequency calculation, and the addition of user configurable virtual channels (e.g. VC1 = CH1 + CH2, VC2 = CH1 * CH2, etc...).

Please note that the full source code is contained on the attached CD ROM, as a full printout would require hundreds of pages. Allow important parts of the source code have been included in this chapter, along with a brief English description on how the code works.

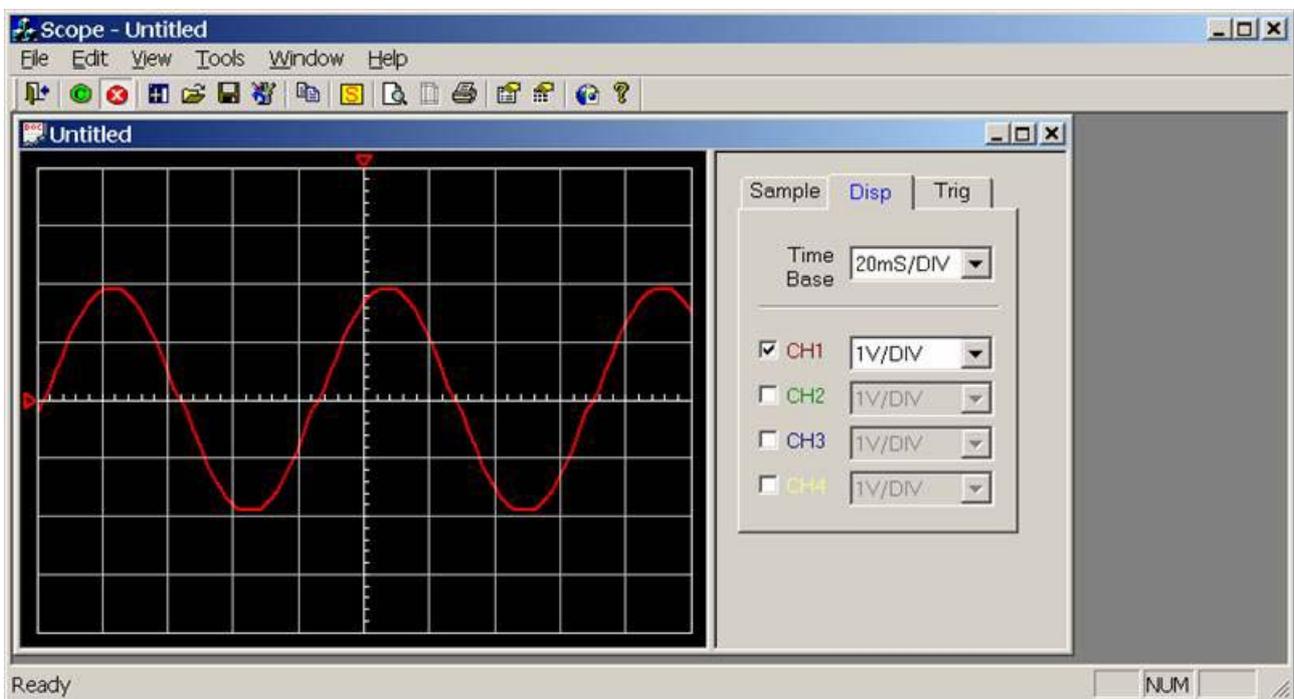


Figure 7.0a. Screen dump of scope.exe (V1.041a, 19/04/2002)

Notice that the display is split into two (see figure 7.0a), the left half displays the waveform and the right half contain the controls. The attached control dialog can be hidden; by right clicking on the waveform display and selecting [Sidebar] or by using the [View] menu (see Figure 7.0c).



Figure 7.0b. Right click menu

At present four channels has been implemented. Data can be received in the real-time mode using the RS232 serial port. The real-time mode communications were first tested using the PIC simulator program, where sinewaves, squarewaves and random waves at variable frequencies and amplitudes were successfully sent to the scope program which correctly displayed the waveforms. (Desktop PC running scope.exe linked to Laptop running sim.exe via RS232 Null mode cable). Once the program was working on the simulator it was tested using a PIC sampling real waveforms, the scope program successfully displayed these waveform correctly, see chapter 11 (testing) for test results at various stages of development.

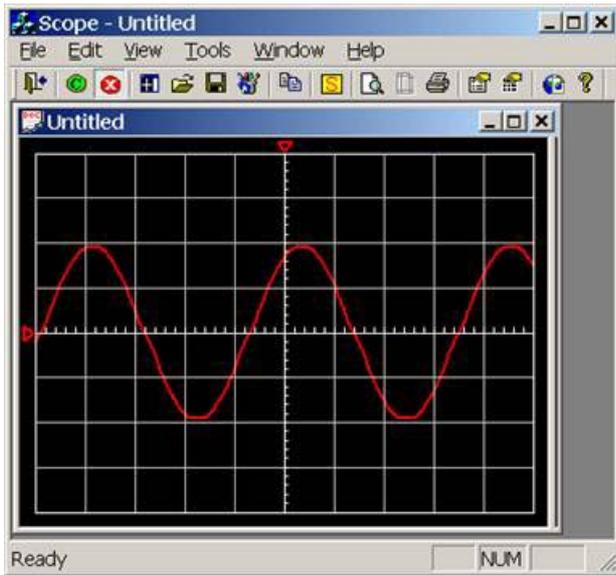


Figure 7.0c. Screen dump of scope.exe with slide bar hidden

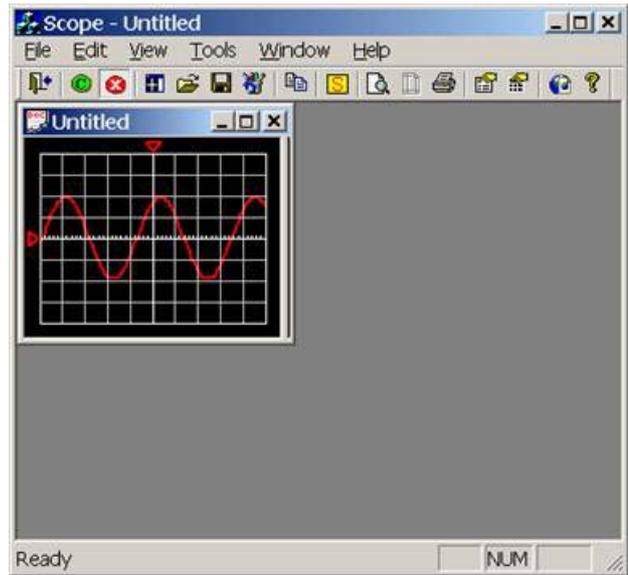


Figure 7.0d. Screen dump demonstrating auto-resize

Notice the small red triangle at the left of the graphical display, this triangle shows the zero point of the displayed waveform. The user using the mouse can click on the triangle and drag it to a new position, hence changing the zero point (see figure 7.0e). Notice if the waveform is too high or too low for the display the top/bottom of the waveform will be cut off.

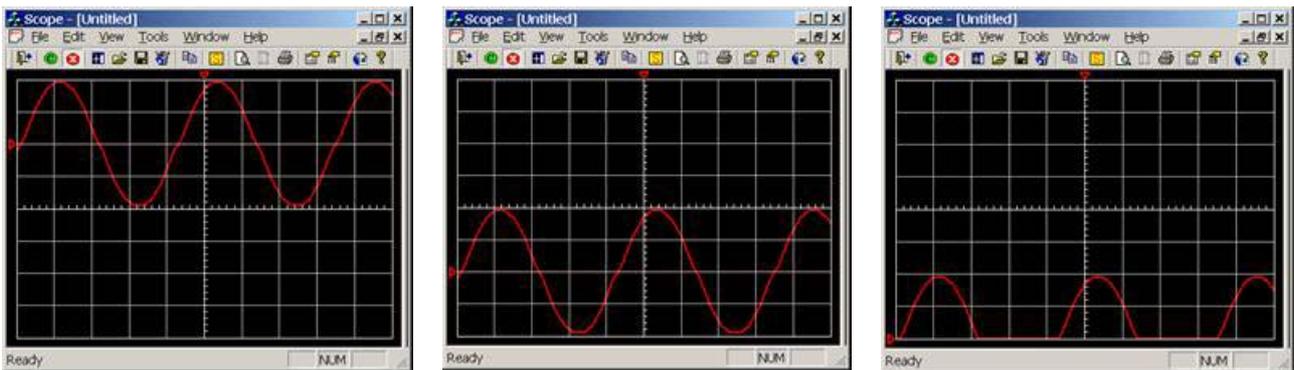


Figure 7.0e. Three screen dumps of scope.exe demonstrating the change of zero point

Notice the small red triangle at the top of the graphical display, this triangle shows the offset position of the displayed waveform (middle of the screen represents zero offset). The user using the mouse can click on the triangle and drag it to a new position, hence changing the waveform offset (see figure 7.0f). Notice that if the waveform is offset to the right, it is possible to see data before the waveform triggered. This feature is mainly useful for helping in the measurement of waveform period time (use in calculation of frequency) where the waveform can be scrolled so that a peak is directly in line with a division line, making accurate measurement easy.

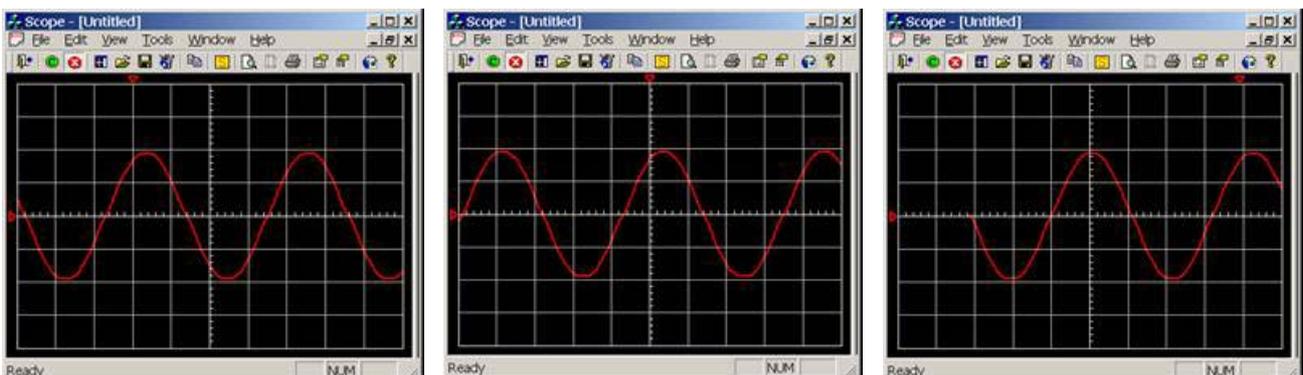


Figure 7.0f. Three screen dumps of scope.exe demonstrating the change of waveform offset

Also notice that the waveform triggers on a low to high edge (software trigger), this is user adjustable in the [Trig] tab of the controls dialog (see figure 7.0g). Notice the user has many options, level triggering means that a trigger occurs when the waveform rises from below to above the user configurable offset level for positive edged triggering and the waveform falls from above to below the user configurable offset level for negative edged triggering. Edge triggering means that the program detects the positive or negative edges of any waveform and at any voltage level, this guarantees that the waveform triggers without the need for user intervention, unlike level mode were the waveform must pass through the selected offset level.

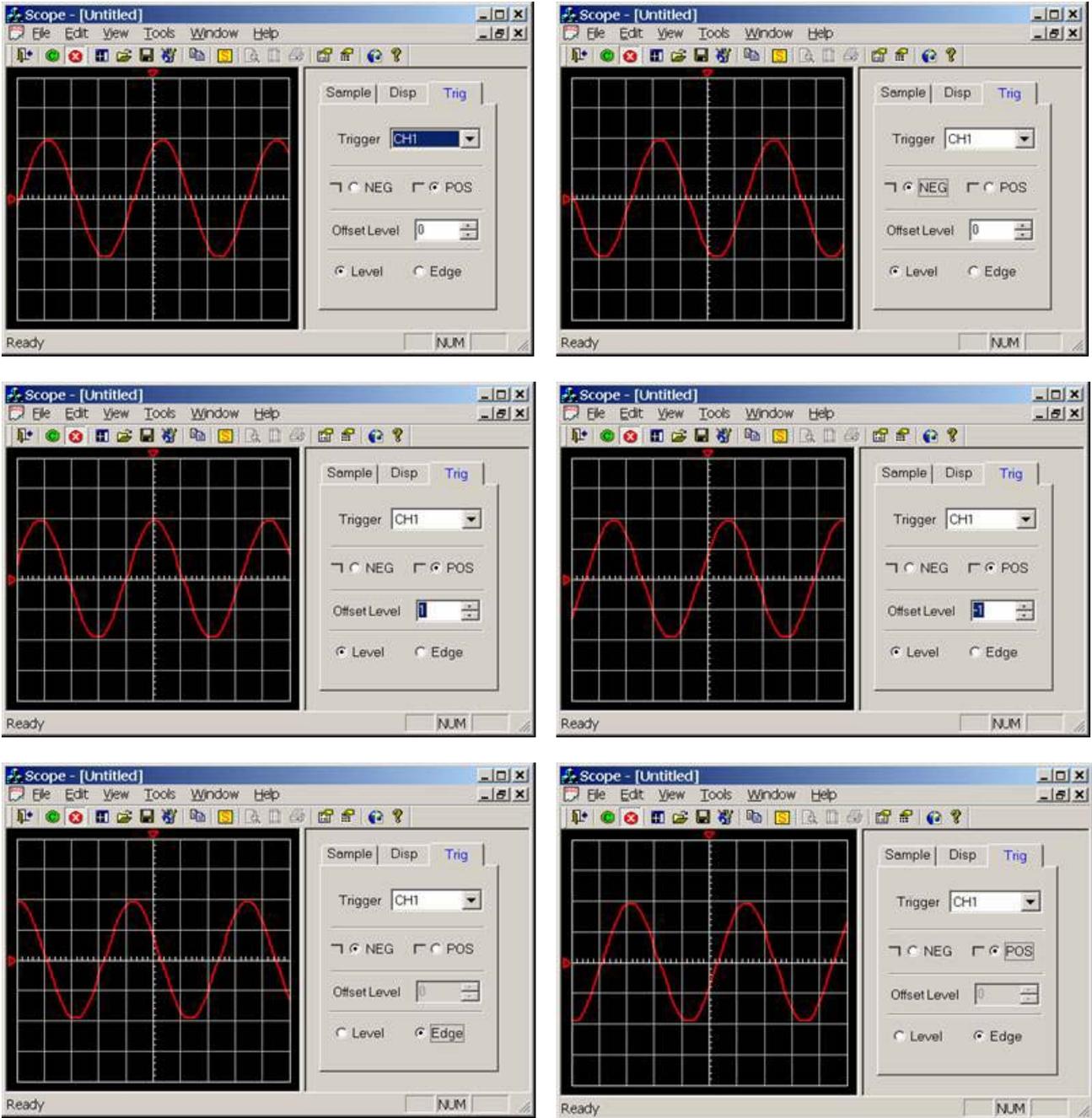


Figure 7.0g. Six screen dumps of scope.exe demonstrating different trigger settings



The user can select which channel to trigger off: CH1, CH2, CH3, or CH4. Notice that there is another option called "IND.", this means independent triggering, where each channel has its one unique trigger point. It is not sure how useful this feature is as generally for dual and quad trace operation waveforms are normally related in some way and timing analysis is normally required. It was decided to include this function for increased program flexibility as this allows for four unrelated waveforms to be monitored simultaneously on the same scope display. Besides it normal practice for

commercial products to have useless features (Bells and Whistles) which will probably never be used in practice, but helps give their product an edge over the competition.

The [Disp] tab of the controls dialog can enable/disable channels simply by clicking the tick box beside each channel, change scope time-base and channel voltage scales (see figure 7.0h). Notice in figure 7.0h there are two enabled channels both of which have zero offset, hence the offset triangle for both channels are in exactly the same place (this is clear due to the colour of the triangle). This is not a problem (e.g. offset of both channels does not change if user clicks on the offset triangle) as channel 1 has absolute priority, hence only channel 1 offset will change separating the channel offsets. Note channel 2 has priority over channel 3 and channel 3 has priority over channel 4. A quick way of moving channel 4 offset for example, if all four channel are enabled and the offset of each is in the same place, is to temporarily disabled channels 1 to 3, move channel 4 offset and enable channels 1 to 3 again.

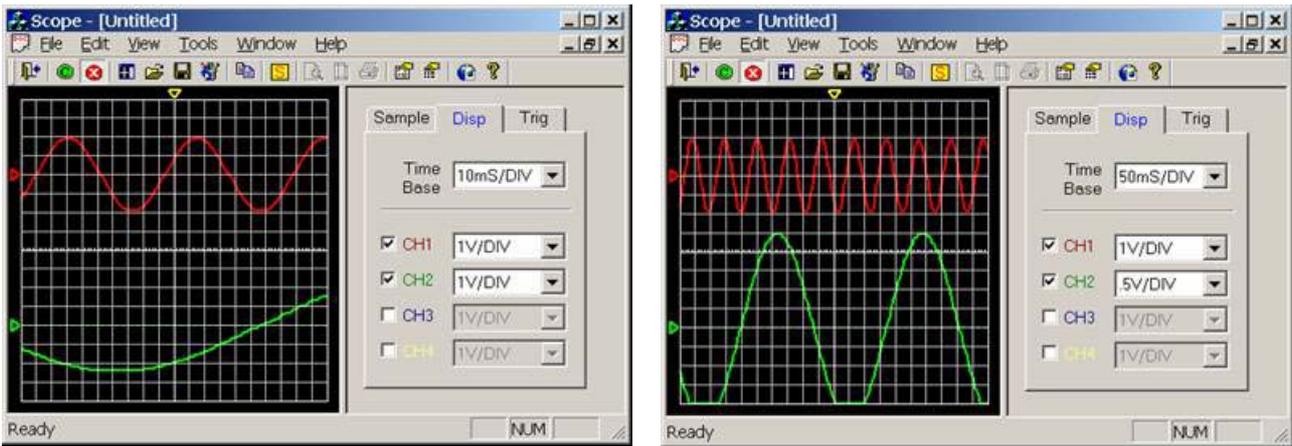


Figure 7.0h. Two screen dumps of scope.exe demonstrating different display settings

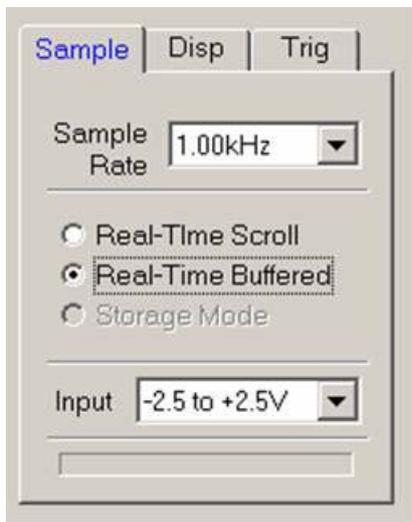


Figure 7.0i. [Sample] Tab of controls dialog

The [Sample] tab of the controls dialog sets the sample rate, mode, and input range of the scope.

Note at present the 'control' communication protocol has not been implemented, hence the selected sample rate will not actually change the rate that the PIC is sampling. But it is important that the correct sample rate is selected as this is used in the calculation of the time-base, else the time-base selected in the [Disp] tab will be incorrect.

Note the input range is selected here, it was expected that this information is sent to the PIC to adjust the gain of the op-amps, but this has not been implement yet. Allow it is important that the correct input range is selected as this changes the binary representation of voltage (e.g. for a range of -2.5 to +2.5V, 0V = -2.5, 512 = 0V, 1024 = 2.5V and for a range of -5 to +5V, -5V = 0, 0V = 512, 5V = 1024).

Sampling mode is also selected here: Real-time scroll mode means that the waveform is sampled using the real-time mode, but stored onto the internal channel arrays in such away that the waveform appears to scroll across the screen (low frequencies only, e.g. useful for ECG monitor). Real-time buffered mode means that data is buffered before copying to the internal channels arrays and is triggered so that the waveform appears to be stable on the display (e.g. useful for periodical waveforms). Storage mode has not been implemented yet, recall that the PIC samples to RAM and then transmits the data to the PC in one large block with added CRC.

The grid configuration dialog box is shown in figure 7.0j; X specifies the number of horizontal squares (1 to 99), Y specifies the number of vertical squares (1 to 99). Note each square has 10 points; hence a maximum resolution of 999 by 999 is possible (assuming Window's desktop resolution is larger than this). The default grid resolution is 10 x 8 (100 x 80), but 20 x 16 (200 x 160) is probably better suited for quad channel operation. Note the higher the grid resolution the more processor hungry the scope program becomes, hence extremely high resolutions are only recommend for high speed modern computers (e.g. 500MHz or above).

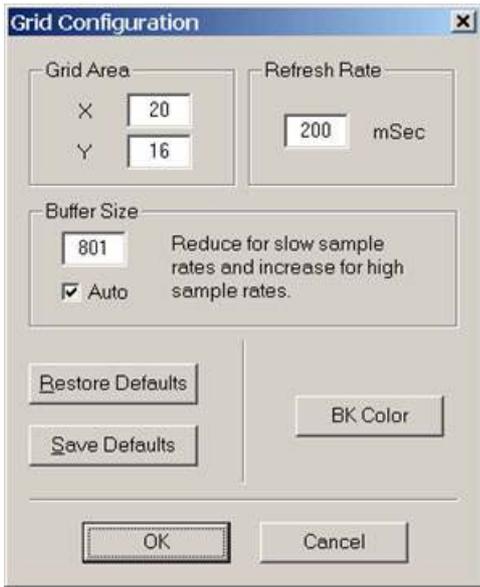


Figure 7.0j. Grid configuration dialog box

The refresh rate specifies the time in milliseconds between screen refreshes. For example 200 milliseconds is the default, that's a refresh rate of $1/0.2 = 5$ Hz. Some traditional analogue engineers who are use working with CRT displays will think that this refresh rate is too low and will result in the screen flickering. This is not the case, unlike CRT displays, the screen holds the current display contains between refreshes, and only redraws the changes made to the waveform when the screen is refreshed (perhaps this should have been called the 'update rate' rather than the 'refresh rate'). The actual refresh rate is that specified by Windows, which for a modern PC is above 100Hz for a CRT display and 50/60Hz for a TFT display.

Note the lower this refresh rate (update rate) the more processor hungry the scope program becomes. A 200 milliseconds update rate will require a PC running at a minimum of 200MHz and at 50 milliseconds update rate will probably require a PC running at least 800MHz to run smoothly, depending on the selected grid resolution and type of video card in the system (hardware acceleration).

The scope program was designed so that multiple scope windows can be displayed; each one with its own selected channels enabled, impendent triggering, time-bases and scales. But it was not sure how useful this feature might be and it may have caused problems during the development process; hence it was considered this feature may be dropped on the finished application. Fortunately this did not cause as many problems as first expected and it was decided to keep this feature, as it may have some use for specialist applications and it's another one of those features ("Bells and Whistles") that sets this project apart from other similar products on the market.

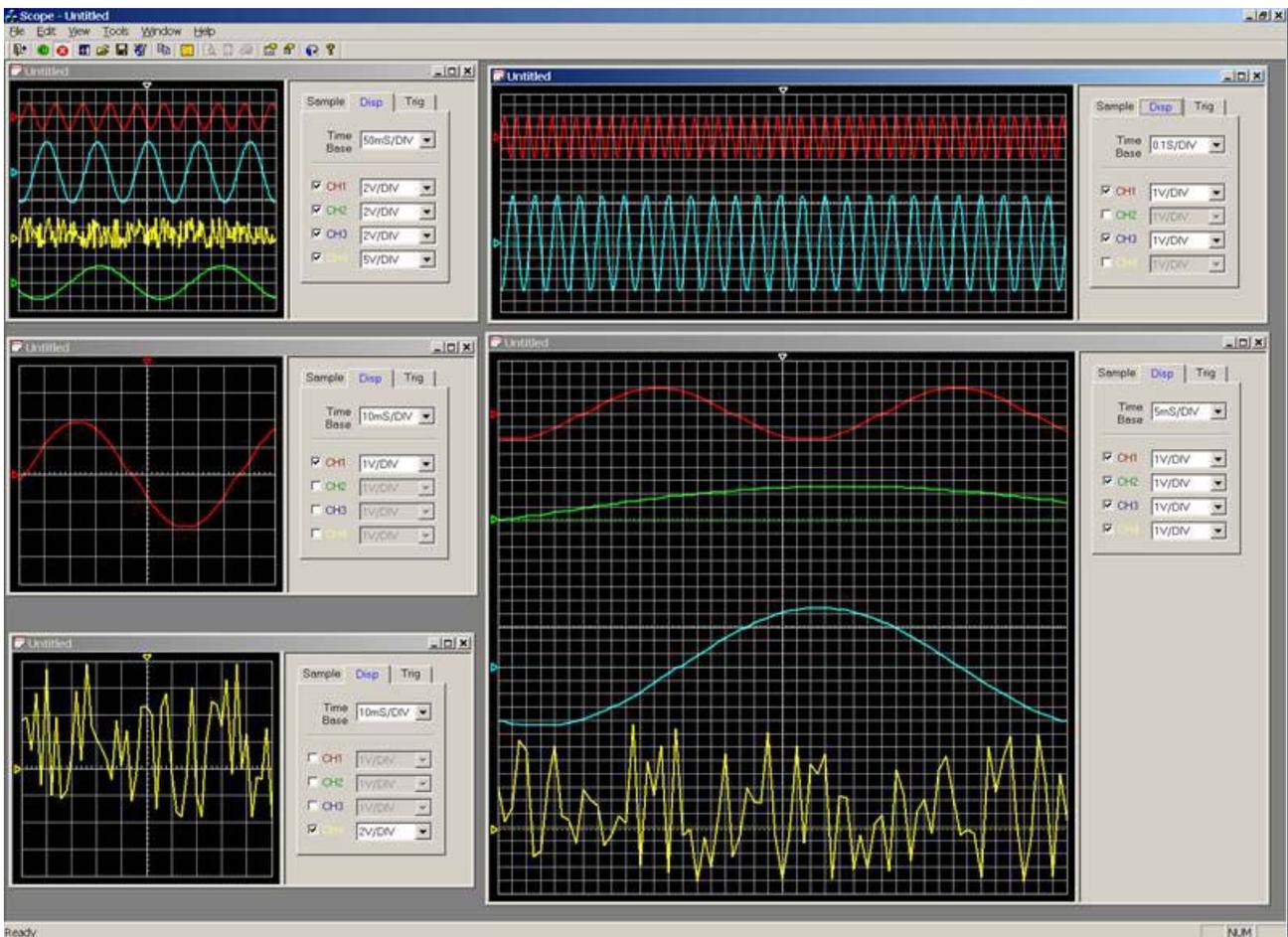


Figure 7.0k. Screen dump scope.exe demonstrating multiple scope views (Window's screen resolution was 1600 x 1200dpi)

Figure 7.0k demonstrates the multiple windows feature, there are 5 views: The top left display, shows all four channels triggering on CH1 at a time-base of 50ms/DIV. The middle left display, shows CH1, with a time-base of 10ms/DIV. The bottom left display, shows CH4, with a 10ms/DIV time-base. The top right display shows CH1 and CH3 at a time-base of 0.1s/DIV. The bottom right display shows all four channels at an extremely high resolution at a time-base of 5ms/DIV and triggering off CH2. The only problem is that all scope windows have the same name "Untitled", clearly this needs to be improved (e.g. "Untitled1", "Untitled2", etc...) as this is confusing to user, the reason for this is because the default automatic naming of the windows was overwritten so that filenames could be placed on the title bar. There is no limit on how many windows that can be opened simultaneously, perhaps until the computer runs out of memory, which on a modern PC could be 1000s of windows.

The scope program is also capable of exporting the scope display to the windows clipboard in the form of a bitmap and exporting the scope display to a bitmap file (*.bmp). The scope program also has its own file format (*.sdf) for saving and opening waveform data and scope settings. Note functionally for printing the scope display directly from the scope program is not fully complete, some additional work is required in this area.

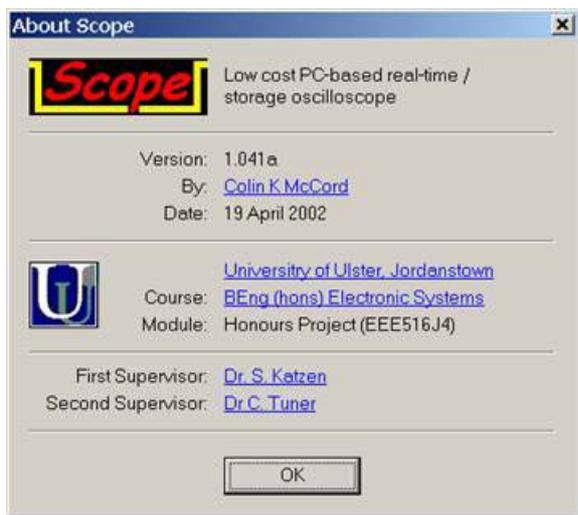


Figure 7.0l. Screen dump of about dialog



Figure 7.0m. Screen dump of about configuration

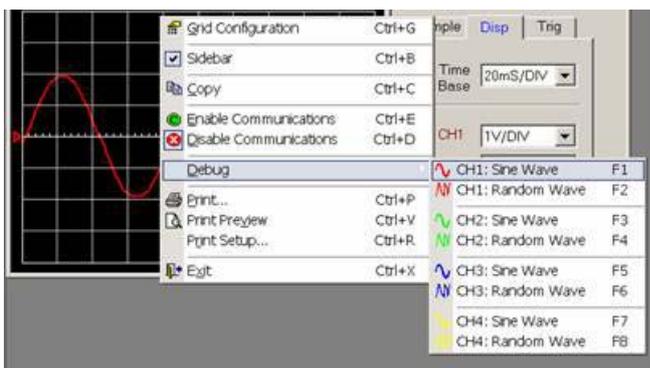


Figure 7.0n. Screen dump of debug menu

The scope program is capable of generating software waveforms (see figure 7.0n), this was used to test the graphical display and triggering methods before the simulation program and PIC hardware was ready for testing. Note this menu will be removed in the final released version.

Another view exists (figure 7.0o), but has not been implemented. It was expected that the user would enter a file name and press [Rec] to stream real-time data to disk and press [Play] to play back recorded real-time data.

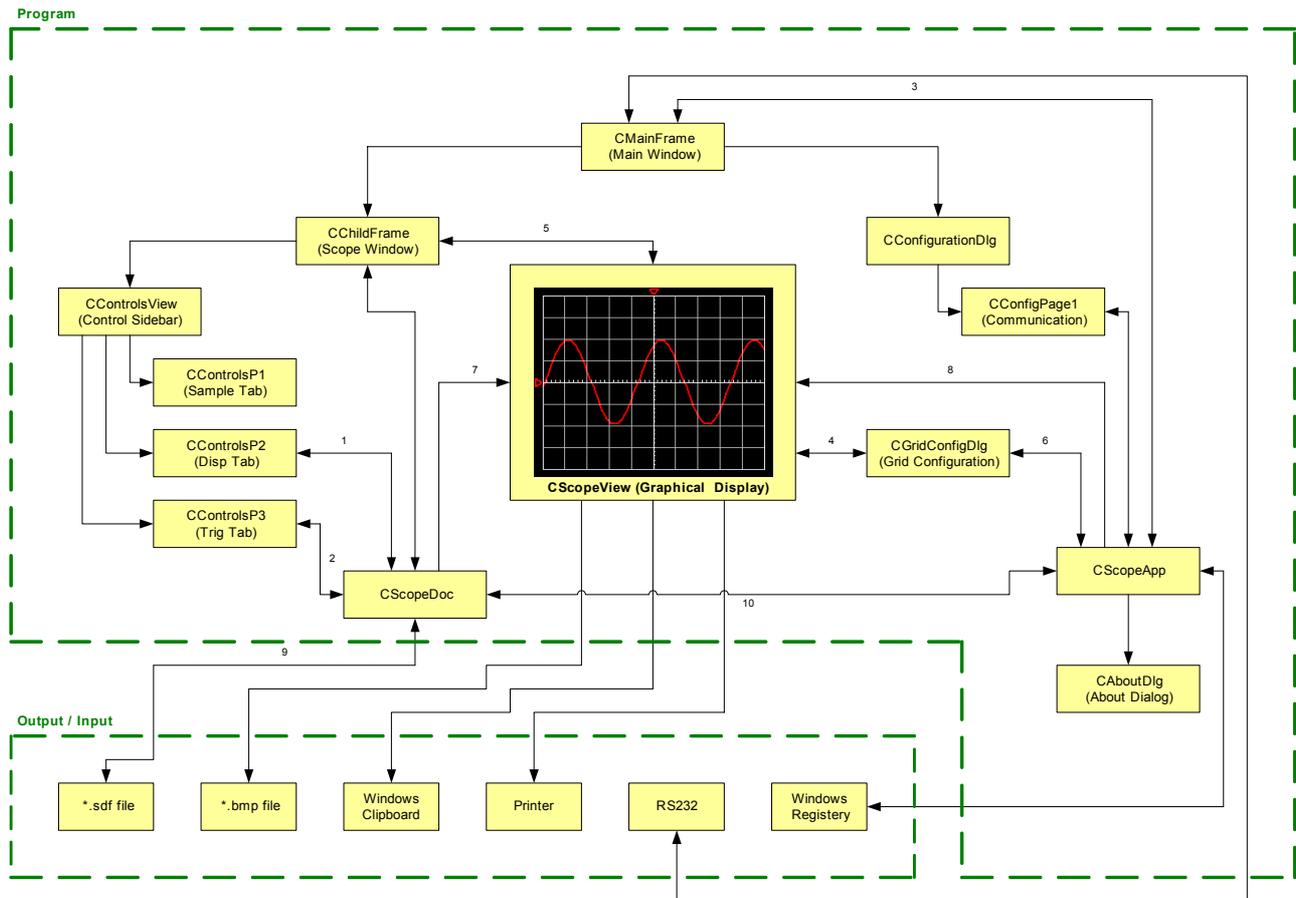


Figure 7.0o. Screen dump of ASCP

When in pause mode the scroll bar labelled 'position' would have been used to manually scroll through the record data.

The tick boxes for channels 1 to 4 were to be used for telling the PIC which channel to sample. The reason why the enable / disable channels (in [Disp] tab) could not be used for this task is because multiple scope windows can be opened with different channels enable / disable hence a global set of enable / disable channels is required to specify which channels to sample.

7.1. Data Flow Diagram



7.1.1. Data Flow Paths

The “Data Flow Diagram” shows how data flows between classes, but it does not give any details on the type of data being transferred. The following is detailed information about data flow in all numbered paths shown on the “Data Flow Diagram”:

Path 1: CControlsP2 ↔ CScopeDoc

```
int m_TimeBaseSelection          int m_nCH1_VOLTS          int m_nCH2_VOLTS
int m_nCH3_VOLTS                int m_nCH4_VOLTS
```

Path 2: CControlsP3 ↔ CScopeDoc

```
int m_nTrigger                  int m_nTriggerOffsetLevel int m_nTriggerType
int m_nTriggerFormat
```

Path 3: CMainFrame ↔ CScopeApp

```
int m_nSampleMode              int m_nBufferSize        int m_nPOS
int m_nCH1Volt[10004]         int m_nCH2Volt[10004]    int m_nCH3Volt[10004]
int m_nCH4Volt[10004]         int m_nInputRange        int m_nSampleRateSelection
```

Path 4: CScopeView ↔ CGridConfigDlg

```
int x                          int y
COLORREF m_ColorGridBackGround; m_nRefreshRate
```

Path 5: ChildFrame ↔ CScopeView

```
m_bShowPanel
```

Path 6: CGridConfigDlg ↔ CScopeApp

```
m_nBufferSize
```

Path 7: CScopeView ← CScopeDoc

```
int m_TimeBaseSelection int          int m_nCH1_VOLTS          int m_nCH2_VOLTS
m_nCH3_VOLTS             int m_nCH4_VOLTS          int m_nTriggerType
int m_nTrigger           int m_nTriggerOffsetLevel int m_nTriggerFormat
bool m_bCH1_ENABLE      bool m_bCH2_ENABLE      bool m_bCH3_ENABLE
bool m_bCH4_ENABLE
```

Path 8: CScopeView ← CScopeApp

```
int m_nCH1Volt[10004]          int m_nCH2Volt[10004]          int m_nCH3Volt[10004]
int m_nCH4Volt[10004]
```

Path 9: CScopeDoc ↔ *.sdf File

```
pApp->m_nCH1Volt[10004]          pApp->m_nCH2Volt[10004]          pApp->m_nCH3Volt[10004]
pApp->m_nCH4Volt[10004]          SView->x                          SView->y
SView->m_nCH1POS                 SView->m_nCH2POS                 SView->m_nCH3POS
SView->m_nCH4POS                 SView->m_nCH1_Offset            SView->m_nCH2_Offset
SView->m_nCH3_Offset             SView->m_nCH4_Offset            pApp->m_nInputRange
pApp->m_nSampleMode              pApp->m_SampleRateSelection      bool m_bCH1_ENABLE
bool m_bCH2_ENABLE              bool m_bCH3_ENABLE              bool m_bCH4_ENABLE
int m_TimeBaseSelection          int m_nCH1_VOLTS                int m_nCH2_VOLTS
int m_nCH3_VOLTS                int m_nCH4_VOLTS                int m_nTrigger
int m_nTriggerType              SView->m_ColorGridBackGround    SView->m_nRefreshRate
```

Path 10: CScopeDoc ↔ CScopeApp

```
int m_nCH1Volt[10004]          int m_nCH2Volt[10004]          int m_nCH3Volt[10004]
int m_nCH4Volt[10004]          int m_nInputRange              int m_nSampleMode
int m_SampleRateSelection
```

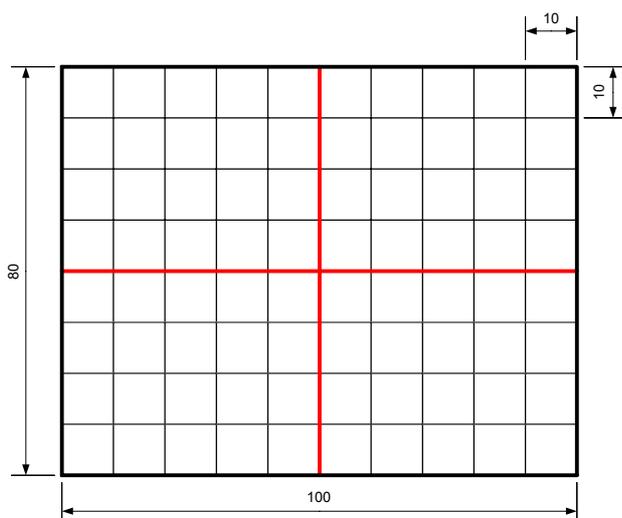
7.2. The Grid

Figure 7.2a. Grid layout

The grid design is shown in figure 7.2a, there are 10 vertical and 10 horizontal points per square. The standard resolution has 10 horizontal squares and 8 vertical squares.

The grid is dynamic, flexible and is one of the key aspects of the scope program.

If the scope window is resized, the grid will automatically resize. The grid resolution is also user configurable; allow the resolution of each square is fixed the number of vertical and horizontal squares are user configurable.

There are two important arrays that must be filled after the scope window has been resized or the grid resolution has been changed (`m_nGridY` & `m_nGridX`), these arrays are used to access individual grid points.

For example the grid layout shown in figure 7.2a, has 100 horizontal (X) points and 80 vertical (Y) points. The scope window size is retrieved from the system and stored in variables `m_nClientWidth` and

$m_nClientHeight$. $m_nClientWidth$ is divided by 100 (X) and $m_nClientHeight$ is divided by 80 (Y), this gives the gap in pixels between each grid point (note variables have been converted to floating point numbers for increased accuracy). The array $m_nGridX[n]$ is filled by looping from $n = 1$ to $n = 100$ (X) and multiplying n by the gap between the horizontal grid points (convert back to data-type int after multiplication has taken place). The array $m_nGridY[n]$ is filled by looping from $n = 1$ to $n = 80$ (Y) and multiplying n by the gap between the vertical grid points (convert back to data-type int after multiplication has taken place). Note there is also a configurable boarder offset (nBD) that must be included in the calculation.

The two arrays can now be used as a coordinate system to access any part of the grid, for example the bottom left edge of the grid is at coordinate $m_nGridX[1]$ by $m_nGridY[1]$ and the top right edge of the grid is at coordinate $m_nGridX[100]$ by $m_nGridY[80]$. Drawing of the grid and waveform traces is achieved using this coordinate system, the main advantage of this is that it does not matter what size the scope display is, the only thing that needs to change is the values contain within the coordinate arrays (extremely flexible).

Function `PerpareGrid()` fills the grid coordinate arrays: -

```
void CScopeView::PrepareGrid()
{
    double fW, fH;
    int n;

    int nBD = 10; /* Border */

    fW = ((double) (m_nClientWidth-(nBD*2)))/((double)x);
    fH = ((double) (m_nClientHeight-(nBD*2)))/((double)y);

    m_nGridX[0] = nBD;
    m_nGridY[0] = nBD;

    for (n = 1; n <=x;n++)
    {
        m_nGridX[n] = ((int) (fW * n))+nBD;
    }

    for (n = 1; n <=y;n++)
    {
        m_nGridY[n] = ((int) (fH * n))+nBD;
    }
}
```

Function `OnDrawGrid()` draws the grid using the grid coordinate system: -

```
void CScopeView::OnDrawGrid()
{
    /* -----
    | Fuction:      void CScopeView::OnDrawGrid() |
    | Description:  Draws grid in memory then updates display. |
    | Return:      Void. |
    | Date:        09/12/2001 |
    | Verison:     1.4 |
    | By:         Colin McCord |
    |----- */
    int n,i;
    CClientDC dc(this);
    CBitmap m_bitmapGrid, *m_bitmapOldGrid = NULL;

    // if we don't have one yet, set up a memory dc for the grid
    if (m_dcGrid.GetSafeHdc() == NULL) m_dcGrid.CreateCompatibleDC(&dc);

    m_bitmapGrid.CreateCompatibleBitmap(&dc, m_nClientWidth, m_nClientHeight);
    m_bitmapOldGrid = m_dcGrid.SelectObject(&m_bitmapGrid);
    m_bitmapOldGrid->DeleteObject(); // Make sure there is not a memory leak

    /* If the pen is not setup, do not continue */
    if(penGrid == NULL) return;

    /* Select the grid pen for use with the DC */
    CPen* pOldPen = m_dcGrid.SelectObject(penGrid);

    CBrush Brush;
    Brush.CreateSolidBrush(m_ColorGridBackGround); // Grid Back Colour
```

```

m_dcGrid.FillRect(&m_ClientRect,&Brush);

/* Draw Border */
m_dcGrid.MoveTo(m_nGridX[0],m_nGridY[0]);
m_dcGrid.LineTo(m_nGridX[x],m_nGridY[0]);
m_dcGrid.LineTo(m_nGridX[x],m_nGridY[y]);
m_dcGrid.LineTo(m_nGridX[0],m_nGridY[y]);
m_dcGrid.LineTo(m_nGridX[0],m_nGridY[0]);

/* Draw V Gray Lines */
for (n=10; n < x;n=n+10)
{
    for(i = m_nGridY[0]; i<m_nGridY[y];i++) m_dcGrid.SetPixel(m_nGridX[n],i,GRAY);
}

/* Draw H grey line */
for (n=10; n < y; n=n+10)
{
    for(i = m_nGridX[0]; i<m_nGridX[x];i++) m_dcGrid.SetPixel(i,m_nGridY[n],GRAY);
}

/* Draw Centre Axis */
m_dcGrid.MoveTo(m_nGridX[x/2],m_nGridY[0]);
m_dcGrid.LineTo(m_nGridX[x/2],m_nGridY[y]);
m_dcGrid.MoveTo(m_nGridX[0],m_nGridY[y/2]);
m_dcGrid.LineTo(m_nGridX[x],m_nGridY[y/2]);

/* Draw V Ticks on H Centre */
for (n=2; n < x; n=n+2)
{
    m_dcGrid.MoveTo(m_nGridX[n],m_nGridY[(y/2)-1]);
    m_dcGrid.LineTo(m_nGridX[n],m_nGridY[(y/2)]);
}

/* Draw H Ticks on V Center */
for (n=2; n < y; n=n+2)
{
    m_dcGrid.MoveTo(m_nGridX[(x/2)],m_nGridY[n]);
    m_dcGrid.LineTo(m_nGridX[(x/2)+1],m_nGridY[n]);
}

m_dcGrid.SelectObject(pOldPen);

/* Update Display */
Invalidate(TRUE);
}

```

7.3. The Traces

The drawing of the traces looks extremely complex when looking at the big picture as channel offsets, zero positions, time-base, sample rate, triggering method, channel disabled/enabled and voltage scales all affect how the scope traces are drawn. Allow the basic principles behind the drawing of the waveforms are simple, so let's try and describe how the waveform traces are drawn step-by-step.

There are four large arrays (one for each channel) in class CScopeApp: -

```

int m_nCH1Volt[10000]; // Voltage * 1000
int m_nCH2Volt[10000]; // Voltage * 1000
int m_nCH3Volt[10000]; // Voltage * 1000
int m_nCH4Volt[10000]; // Voltage * 1000

```

These arrays are global and can be accessed from any part of the program. Sampled readings for each channel are stored in these arrays; filled by the communication protocol, note the method used to fill these arrays is depended on which mode is selected e.g. for scroll mode every location is moved one placed to the left and the new reading is placed at the end of the array. Note the 10-bit ADC readings have already been converted into voltage depending on the selected input voltage range and multiplied by 1000 to increase resolution without having to use floating point numbers (e.g. for range -10 to 10V: -10V = -10000, 0V = 0 and 10V = 10000).

The basic principle is that the program continuously scans through these arrays updating the display at a user configurable refresh rate.

Software interrupt timer for redrawing traces: -

```
void CScopeView::OnTimer(UINT nIDEvent)
{
    if (nIDEvent == nTimerRefresh)
    {
        KillTimer(nTimerRefresh);

        Trigger();    /** Cal Trigger Points */

        /* Set scales */
        m_nVPD_CH1 = pDoc->m_nCH1_VOLTS;
        m_nVPD_CH2 = pDoc->m_nCH2_VOLTS;
        m_nVPD_CH3 = pDoc->m_nCH3_VOLTS;
        m_nVPD_CH4 = pDoc->m_nCH4_VOLTS;

        /** Draw Traces */
        OnDrawCH1();
        OnDrawCH2();
        OnDrawCH3();
        OnDrawCH4();

        /* Enable Timer again */
        SetTimer(nTimerRefresh,m_nRefreshRate,NULL);
    }
    CView::OnTimer(nIDEvent);
}
```

The first thing to note is that the timer is stopped while executing the OnTimer() function and started again once the redraw is finished. The reason for this is to reduce the chance of the program locking up because of accumulating delays caused by the refresh rate selected by the user being shorter than the time required to redraw the display. In reality the true refresh rate is not the rate specified by the user, but the time taken to redraw the display plus the rate specified by the user, this insures that the program does not lockup especially when running on slow PCs (<200 MHz).

Function trigger() is called to find the trigger point for all channels, then the channel voltages scales are set to that selected by the combo box, then the channel traces are drawn.

This function calculates the trigger point: -

```
void CScopeView::Trigger()
{
    int a = pDoc->m_nTriggerOffsetLevel;

    /* CH1 Tigger */
    for(int i = 1;i <= 1000;i++)
    {
        if (pDoc->m_nTriggerType == 0)
        {
            if(pDoc->m_nTriggerFormat == 1) a = pApp->m_nCH1Volt[i-1]; // Edge Triggered

            /* trigger POS to NEG */
            if (pApp->m_nCH1Volt[i] >= a && pApp->m_nCH1Volt[i+1] < a)
            {
                m_nTriggerCH1 = i;

                /* Disable trigger for scroll mode */
                if (pApp->m_nSampleMode == 0) m_nTriggerCH1 = 0; // disbale trigger

                i = 1001;
                break;
            }
        }
        else
        {
            if(pDoc->m_nTriggerFormat == 1) a = pApp->m_nCH1Volt[i-1]; // Edge Triggered

            /* trigger NEG to POS */
            if (pApp->m_nCH1Volt[i] <= a && pApp->m_nCH1Volt[i+1] > a)
            {

```

```

        m_nTriggerCH1 = i;

        /* Disable trigger for scroll mode */
        if (pApp->m_nSampleMode == 0) m_nTriggerCH1 = 0; // disable trigger

        i = 1001;
        break;
    }
}

...
NOTE channel 2-4 trigger point calculated the same way as channel 1
...

switch (pDoc->m_nTrigger)
{
    case 0: // trig on CH1
        m_nTriggerCH2 = m_nTriggerCH1;
        m_nTriggerCH3 = m_nTriggerCH1;
        m_nTriggerCH4 = m_nTriggerCH1;
        break;
    case 1: // trig on CH2
        m_nTriggerCH1 = m_nTriggerCH2;
        m_nTriggerCH3 = m_nTriggerCH2;
        m_nTriggerCH4 = m_nTriggerCH2;
        break;
    case 2: // trig on CH3
        m_nTriggerCH1 = m_nTriggerCH3;
        m_nTriggerCH2 = m_nTriggerCH3;
        m_nTriggerCH4 = m_nTriggerCH3;
        break;
    case 3: // trig on CH4
        m_nTriggerCH1 = m_nTriggerCH4;
        m_nTriggerCH2 = m_nTriggerCH4;
        m_nTriggerCH3 = m_nTriggerCH4;
        break;
}
}

```

Detection of the trigger point is simple; basically the program scans the array and looks for a high-to-low or a low-to-high edge. The location of which is stored (m_nTriggerCH1..4), this producer is carried out for all four channels. If a specific channel is selected as the trigger, the trigger points of the other channels are made equal to the trigger point of the selected channel.

A memory DCs and pen is required for each trace: -

```

CPen* penCH1;
CDC  m_dcCH1;

CDC  m_dcCH2;
CPen* penCH2;

CDC  m_dcCH3;
CPen* penCH3;

CDC  m_dcCH4;
CPen* penCH4;

```

Variables are initialised in the CScopeView constructor: -

```

CScopeView::CScopeView()
{
    m_bitmapCH1 = NULL;
    m_bitmapCH2 = NULL;
    m_bitmapCH3 = NULL;
    m_bitmapCH4 = NULL;

    m_nClientHeight= 0;
    m_nClientWidth = 0;
    x = 100;
    y = 80;

    /** Zero Point Positions **/

```

```

    m_nCH1POS = 40;
    m_nCH2POS = 20;
    m_nCH3POS = 50;
    m_nCH4POS = 60;

    m_nCH1_Offset = 0;
    m_nCH2_Offset = 0;
    m_nCH3_Offset = 0;
    m_nCH4_Offset = 0;

    m_nTriggerCH1 = 0;
    m_nVPD_CH1 = 100; /* 1 voltage per division */
    m_nVPD_CH2 = 100;
    m_nVPD_CH3 = 100;
    m_nVPD_CH4 = 100;

    penGrid =NULL;
    penCH1 = NULL;
    penCH2 = NULL;
    penCH3 = NULL;
    penCH4 = NULL;

    m_bChangeZeroCH1 = false;
    m_bChangeZeroCH2 = false;
    m_bChangeZeroCH3 = false;
    m_bChangeZeroCH4 = false;

    m_bChangeOffsetCH1 = false;

    m_nRefreshRate = 200; // Default refresh rate 200ms
}

```

Refresh timer, trace colours, grid colour and the grid background colour are setup here: -

```

void CScopeView::OnInitialUpdate()
{
    CView::OnInitialUpdate();
    pDoc = (CScopeDoc*)m_pDocument;
    pApp = (CScopeApp*)AfxGetApp();

    SetTimer(nTimerRefresh,m_nRefreshRate,NULL);

    penGrid = new(CPen);
    penGrid->CreatePen(PS_SOLID,1,WHITE);
    penCH1 = new(CPen);
    penCH1->CreatePen(PS_SOLID,2,LIGHTRED);
    penCH2 = new(CPen);
    penCH2->CreatePen(PS_SOLID,2,LIGHTGREEN);
    penCH3 = new(CPen);
    penCH3->CreatePen(PS_SOLID,2,LIGHTBLUE);
    penCH4 = new(CPen);
    penCH4->CreatePen(PS_SOLID,2,YELLOW);

    m_ColorGridBackGround = BLACK;

    OnDrawGrid();
}

```

This function is called when a channel voltage scale combo box is changed: -

```

void CControlsP2::OnChange()
{
    CString temp;

    /** Setup Channel 1 Volts per Dev **/
    GetDlgItemText(IDC_CH1_VOLT, temp);
    if (temp == _T("10mV/DIV")) pDoc->m_nCH1_VOLTS = 1;
    if (temp == _T("20mV/DIV")) pDoc->m_nCH1_VOLTS = 2;
    if (temp == _T("50mV/DIV")) pDoc->m_nCH1_VOLTS = 5;
    if (temp == _T(".1V/DIV")) pDoc->m_nCH1_VOLTS = 10;
    if (temp == _T(".2V/DIV")) pDoc->m_nCH1_VOLTS = 20;
    if (temp == _T(".5V/DIV")) pDoc->m_nCH1_VOLTS = 50;
    if (temp == _T("1V/DIV")) pDoc->m_nCH1_VOLTS = 100;
    if (temp == _T("2V/DIV")) pDoc->m_nCH1_VOLTS = 200;
    if (temp == _T("5V/DIV")) pDoc->m_nCH1_VOLTS = 500;
    if (temp == _T("10V/DIV")) pDoc->m_nCH1_VOLTS = 1000;
    if (temp == _T("20V/DIV")) pDoc->m_nCH1_VOLTS = 2000;
}

```

```

if (temp == _T("50V/DIV")) pDoc->m_nCH1_VOLTS = 5000;

...
Channel 2-4 done in exactly the same way as channel 1
...
}

```

This function prepares the traces: -

```

void CScopeView::PrepareTraces()
{
/* -----
| Fuction:      void CScopeView::PrepareTraces()      |
| Description:  Creates trace memory DCs and bitmaps. |
| Return:      Void.                                  |
| Date:        15/03/2002                             |
| Verison:     1.1                                     |
| By:         Colin McCord                             |
|-----*/

    CClientDC dc(this);    // Screen DC.

    /*** Channel 1 ***/
    CBitmap *m_bitmapOldCH1 = NULL;    // Pointer to old display bitmap

    m_bitmapCH1 = new CBitmap;    // Create a new bitmap

    /** Create new bitmap and delete old, (Channel 1) **/
    if (m_dcCH1.GetSafeHdc() == NULL) m_dcCH1.CreateCompatibleDC(&dc);

    m_bitmapCH1->CreateCompatibleBitmap(&dc, m_nClientWidth, m_nClientHeight);
    m_bitmapOldCH1 = m_dcCH1.SelectObject(m_bitmapCH1);
    m_bitmapOldCH1->DeleteObject(); // Make Sure there is not a memory leak

    ...
    Channels 2-4 prepared in exactly the same way as channel 1
    ...
}

```

This function draws Channel 1: -

```

void CScopeView::OnDrawCH1()
{
/* -----
| Fuction:      void CScopeView::OnDrawCH1()          |
| Description:  Draws CH1 waveform, in memory then updates |
|              display.                                |
| Return:      Void.                                  |
| Date:        04/04/2002                             |
| Verison:     1.9                                     |
| By:         Colin McCord                             |
|-----*/

    CClientDC dc(this);
    int i,n,y1,y2,a=1,m_Skip, m_MUX;
    CBrush Brush;
    Brush.CreateSolidBrush(BLACK); // Set brush colour to black
    CRect CRectTEMP;

    // Channel 1 Member DC not setup yet
    if (m_dcCH1.GetSafeHdc() == NULL)
    {
        PrepareTraces();
    }

    /* If channel 1 pen is not created yet, do not continue */
    if (penCH1 == NULL) return;

    /* Select CH1 pen */
    CPen* pOldPen = m_dcCH1.SelectObject(penCH1);

    /* Clear display*/
    m_dcCH1.SetBkColor(BLACK);
    m_dcCH1.FillRect(&m_ClientRect, &Brush);
}

```

```

/* If Channel 1 is disabled do not continue */
if (pDoc->m_bCH1_ENABLE == false)
{
    m_dcCH1.SelectObject(pOldPen);
    Invalidate(TRUE); // Redraw Display
    return;
}

if (pDoc->m_TimeBaseMUX < 0) // Increase Timebase
{
    m_Skip = pDoc->m_TimeBaseMUX * -1;
    m_MUX = 1;
}
else // Reduce Timebase
{
    m_Skip = 1;
    m_MUX = pDoc->m_TimeBaseMUX;
}

/* Draw waveform to memory DC */
for(i = m_nTriggerCH1-(m_nCH1_Offset*m_MUX/m_Skip); i <= 10000; i = i+m_MUX)
{
    if(m_Skip >5000) break; // Check for possible overflow

    y1 = m_nCH1POS+pApp->m_nCH1Volt[i-m_MUX]/m_nVPD_CH1;
    if (y1 > y) y1 = y;
    if (y1 < 0) y1 = 0;
    if (a-m_Skip<0) a = m_Skip;
    if (i >=0) m_dcCH1.MoveTo(m_nGridX[a-m_Skip],m_nGridY[y1]);

    y2 = m_nCH1POS+pApp->m_nCH1Volt[i]/m_nVPD_CH1;
    if (y2 > y) y2 = y;
    if (y2 < 0) y2 = 0;
    if (i >=0) m_dcCH1.LineTo(m_nGridX[a],m_nGridY[y2]);

    a = a + m_Skip;

    if ( a > x)
    {
        i = 10001;
        break;
    }
}

/** draw zero point Arrow */
m_dcCH1.MoveTo(8,m_nGridY[m_nCH1POS]);
m_dcCH1.LineTo(2,m_nGridY[m_nCH1POS]-5);
m_dcCH1.LineTo(2,m_nGridY[m_nCH1POS]+5);
m_dcCH1.LineTo(8,m_nGridY[m_nCH1POS]);

/* Draw Zero point dotted line */
for(i = 10; i<m_nGridX[x]-3;i+=6)
{
    for(n=0;n<=3;n++) m_dcCH1.SetPixel(i+n,m_nGridY[m_nCH1POS],LIGHTRED);
}

/** draw offset Arrow */
m_dcCH1.MoveTo(m_nGridX[x/2+m_nCH1_Offset],8);
m_dcCH1.LineTo(m_nGridX[x/2+m_nCH1_Offset]-5,2);
m_dcCH1.LineTo(m_nGridX[x/2+m_nCH1_Offset]+5,2);
m_dcCH1.LineTo(m_nGridX[x/2+m_nCH1_Offset],8);

m_dcCH1.SelectObject(pOldPen);
Invalidate(TRUE); // Redraw Display
}

```

The code marked in **red** is the actual drawing of the waveform; basically the array is scanned and drawn to the channel DC (m_dcCH1) using the grid coordinate arrays m_nGridX[] and m_nGridY[] . The code looks more complex because of the following variables: m_nTriggerCH1 specifies the array trigger offset, m_nCH1_Offset specifies the channel offset, m_Skip specifies the number of array locations to skip to achieve the required time-base, m_MUX specifies the number of array location to duplicate to achieve the required time-base, and m_nVPD_CH1 sets the voltage scale. Note drawing of the zero point and offset arrows are also drawn in this function and all four channels are drawn in exactly the same way.

This function is called by the system when the scope display is resized: -

```
void CScopeView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    m_nClientWidth = cx;
    m_nClientHeight = cy;

    m_ClientRect.bottom = 0;
    m_ClientRect.left = 0;
    m_ClientRect.right = m_nClientWidth;
    m_ClientRect.top = m_nClientHeight;

    PrepareGrid();
    PrepareTraces();
    OnDrawGrid();
    OnDrawCH1();
    OnDrawCH2();
    OnDrawCH3();
    OnDrawCH4();
}
```

Function OnSize() first retrieves the new size of the scope display from the system and updates the m_nClientRect variable. Then it calls function PrepareGrid() to recalculate and fill the grid coordinate arrays. Then prepares the traces, redraws the grid and all of the traces.

7.4. Storing and Restoring the Previous Position of the Main Frame.

Basically the window size and position is saved to the windows registry before the application closes and restored when the program starts up.

Two functions used for saving and restoring windows position: -

```
////////////////////////////////////
// saving/restoring window state

static TCHAR BASED_CODE szSection[] = _T("Settings");
static TCHAR BASED_CODE szWindowPos[] = _T("WindowPos");
static TCHAR szFormat[] = _T("%u,%u,%d,%d,%d,%d,%d,%d,%d,%d");

static BOOL PASCAL NEAR ReadWindowPlacement(LPWINDOWPLACEMENT pwp)
{
    CString strBuffer = AfxGetApp()->GetProfileString(szSection, szWindowPos);
    if (strBuffer.IsEmpty())
        return FALSE;

    WINDOWPLACEMENT wp;
    int nRead = _stscanf(strBuffer, szFormat,
        &wp.flags, &wp.showCmd,
        &wp.ptMinPosition.x, &wp.ptMinPosition.y,
        &wp.ptMaxPosition.x, &wp.ptMaxPosition.y,
        &wp.rcNormalPosition.left, &wp.rcNormalPosition.top,
        &wp.rcNormalPosition.right, &wp.rcNormalPosition.bottom);

    if (nRead != 10)
        return FALSE;

    wp.length = sizeof wp;
    *pwp = wp;
    return TRUE;
}

static void PASCAL NEAR WriteWindowPlacement(LPWINDOWPLACEMENT pwp)
// write a window placement to settings section of app's registry
{
    TCHAR szBuffer[sizeof("-32767")*8 + sizeof("65535")*2];

    wsprintf(szBuffer, szFormat,
        pwp->flags, pwp->showCmd,
        pwp->ptMinPosition.x, pwp->ptMinPosition.y,
        pwp->ptMaxPosition.x, pwp->ptMaxPosition.y,
        pwp->rcNormalPosition.left, pwp->rcNormalPosition.top,
```

```

        pwp->rcNormalPosition.right, pwp->rcNormalPosition.bottom);
    AfxGetApp()->WriteProfileString(szSection, szWindowPos, szBuffer);
}

////////////////////////////////////

```

This function restores window position from windows registry: -

```

void CMainFrame::InitialShowWindow(UINT nCmdShow)
{
    /** Restore window position from registry, by CKM 05/02/2002 **/
    WINDOWPLACEMENT wp;
    if (!ReadWindowPlacement(&wp))
    {
        ShowWindow(nCmdShow);
        return;
    }
    if (nCmdShow != SW_SHOWNORMAL) wp.showCmd = nCmdShow;
    SetWindowPlacement(&wp);
    ShowWindow(wp.showCmd);
}

```

Function CMainFrame::InitialShowWindow() is called from CScopeApp::InitInstance(): -

```

BOOL CScopeApp::InitInstance()
{
    ...

    // The main window has been initialized, so show and update it.
    pMainFrame->InitialShowWindow(m_nCmdShow);
    pMainFrame->UpdateWindow();

    return TRUE;
}

```

Before the window is destroyed its position is saved in the windows registry: -

```

void CMainFrame::OnClose()
{
    // before mainframe it is destroyed, save the position of the window
    WINDOWPLACEMENT wp;
    wp.length = sizeof wp;
    if (GetWindowPlacement(&wp))
    {
        wp.flags = 0;
        if (IsZoomed())
            wp.flags |= WPF_RESTORETOMAXIMIZED;
        // and write it to the windows registry
        WriteWindowPlacement(&wp);
    }

    CMDIFrameWnd::OnClose();
}

```

Note the same technique can be used for saving and restoring the position of the scope display window, but at present multiple scope display windows can be used and it's not ideal if they all open at exactly the same place.

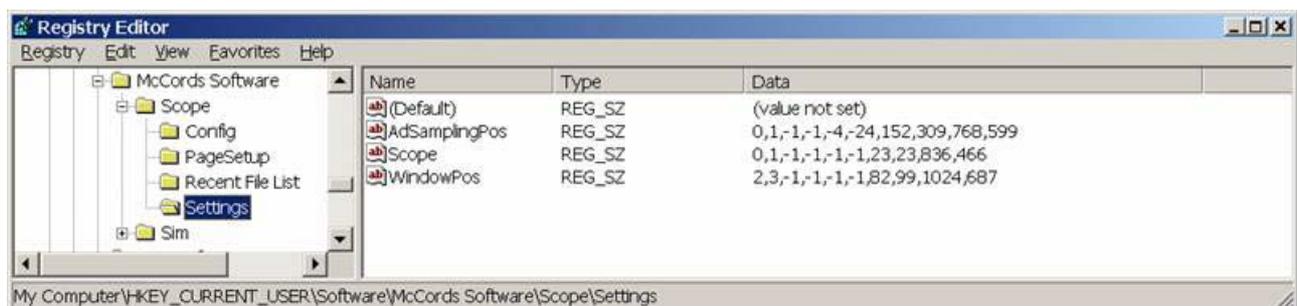


Figure 7.4a. Screen dump of window registry showing where the window position is stored

7.5. Setting-Up RS232 Communications

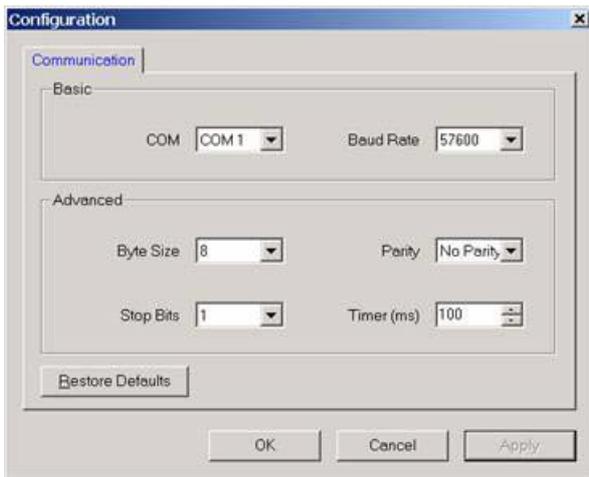


Figure 7.5a. Screen dump of the configuration dialog box

The configuration settings are easily configurable using the configuration dialog from menu [Tools] [Configuration]. This dialog is user friendly and easy to use, notice that a tab control was used and at present there is only one tab (communication), additional tabs will be added at a later date (e.g. calibration). Note when the user hits [OK] or [Apply] the configuration settings are saved to the windows registry, [Apply] button is only enabled if changes have been made, and the [Restore Defaults] button is only enabled when the current configuration differs from the defaults, see figure 7.5a.

When user hits [OK] or [Apply] this function is called: -

```
void CConfigPage1::Save_Changes ()
{
    UpdateData (TRUE);
    CScopeApp* pApp = (CScopeApp*)AfxGetApp();
    pApp->WriteProfileInt("Config","COM",m_Combo_COM.GetCurSel());
    pApp->WriteProfileInt("Config","Baud",m_Combo_Baud.GetCurSel());
    pApp->WriteProfileInt("Config","Byte Size",m_Combo_Byte_Size.GetCurSel());
    pApp->WriteProfileInt("Config","Parity",m_Combo_Parity.GetCurSel());
    pApp->WriteProfileInt("Config","Stop Bits",m_Combo_Stop_Bits.GetCurSel());
    pApp->WriteProfileInt("Config","Timer",GetDlgItemInt(IDC_TIMER));

    OnChange();
}
```

Screen dump showing where RS232 configuration is stored: -

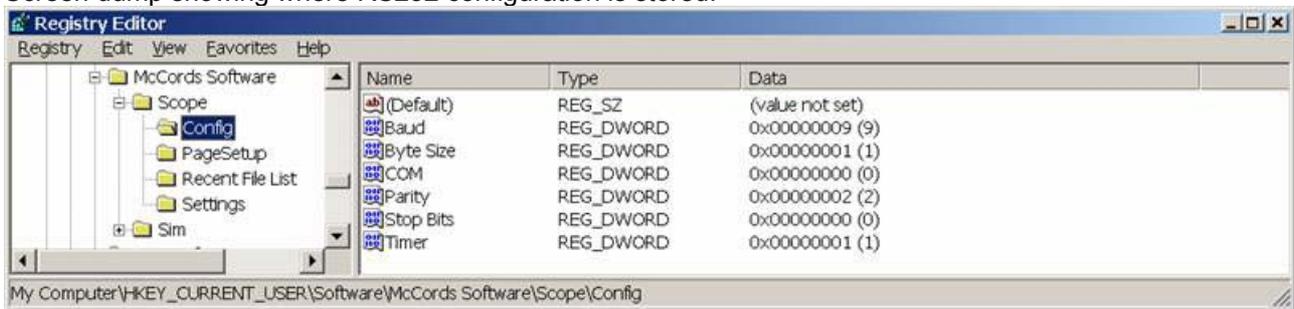


Figure 7.5b. Screen dump of windows registry

Note direct access to the hardware is not allowed under Windows NT/2000/XP. Interaction with the serial port is achieved through a file handle and various WIN32 communication API's. This method is Windows 95/98/ME compatible.

This function initialises the communications using configuration settings from windows registry: -

```
void CMainFrame::InitialCom()
{
    /* 24/12/01 by colin mccord */

    CScopeApp* pApp = (CScopeApp*)AfxGetApp();

    int temp_int;
    DWORD baud;
    BYTE parity, byte, StopBits;

    temp_int = pApp->GetProfileInt("Config","COM",D_COM);
    if(temp_int == 0)        m_sComPort = "Com1";
```

```

else if(temp_int == 1)      m_sComPort = "Com2";
else if(temp_int == 2)      m_sComPort = "Com3";
else if(temp_int == 3)      m_sComPort = "Com4";
else                          m_sComPort = "Com1";

temp_int = pApp->GetProfileInt("Config","Baud",D_BAUD);
if (temp_int == 0)          baud = 1200;
else if(temp_int == 1)      baud = 2400;
else if(temp_int == 2)      baud = 4800;
else if(temp_int == 3)      baud = 9600;
else if(temp_int == 4)      baud = 14400;
else if(temp_int == 5)      baud = 19200;
else if(temp_int == 6)      baud = 32768;
else if(temp_int == 7)      baud = 38400;
else if(temp_int == 8)      baud = 57600;
else if(temp_int == 9)      baud = 115200;
else baud = 9600;

temp_int = pApp->GetProfileInt("Config","Byte Size",D_BYTE);
if(temp_int == 0)           byte = 7;
else if(temp_int == 1)      byte = 8;
else                          byte = 8;

temp_int = pApp->GetProfileInt("Config","Parity",D_PARITY);
if(temp_int == 0)           parity = EVENPARITY;
else if(temp_int == 1)      parity = MARKPARITY;
else if(temp_int == 2)      parity = NOPARITY;
else if(temp_int == 3)      parity = ODDPARITY;
else if(temp_int == 4)      parity = SPACEPARITY;
else parity = NOPARITY;

temp_int = pApp->GetProfileInt("Config","Stop Bits",D_STOP);
if(temp_int == 0)           StopBits = ONESTOPBIT;
else if(temp_int == 1)      StopBits = ONE5STOPBITS;
else if(temp_int == 2)      StopBits = TWOSTOPBITS;
else                          StopBits = ONESTOPBIT;

m_hCom = CreateFile(m_sComPort,
    GENERIC_READ | GENERIC_WRITE,
    0, // exclusive access
    NULL, // no security
    OPEN_EXISTING,
    0, // no overlapped I/O
    NULL); // null template

// Check the returned handle for INVALID_HANDLE_VALUE and then
// set the buffer sizes.

m_bPortReady = SetupComm(m_hCom, 1000, 1000); // set buffer sizes

m_bPortReady = GetCommState(m_hCom, &m_dcb);
m_dcb.BaudRate = baud;
m_dcb.ByteSize = byte;
m_dcb.Parity = parity;
m_dcb.StopBits = ONESTOPBIT;

m_bPortReady = SetCommState(m_hCom, &m_dcb);

m_bPortReady = GetCommTimeouts (m_hCom, &m_CommTimeouts);

m_CommTimeouts.ReadIntervalTimeout = 3;
m_CommTimeouts.ReadTotalTimeoutConstant = 3;
m_CommTimeouts.ReadTotalTimeoutMultiplier = 3;
m_CommTimeouts.WriteTotalTimeoutConstant = 10;
m_CommTimeouts.WriteTotalTimeoutMultiplier = 10;

m_bPortReady = SetCommTimeouts (m_hCom, &m_CommTimeouts);
}

```

This function is used to read one character from the RS232 serial port: -

```

bool CMainFrame::readchr(unsigned char *c)
{

```

```

    unsigned char  Buffer[128];
    unsigned long  a = 0;

    bReadRC = ReadFile(m_hCom, &Buffer, 1, &a, NULL);

    if (a == 0) return false;    // No char received
    else
    {
        *c = Buffer[0];
        return true;
    }
}

```

This function is used to transmit one character to the RS232 serial port: -

```

void CMainFrame::trans(char c)
{
    char buffer[2];
    buffer[0] = c;
    bWriteRC = WriteFile (m_hCom, buffer, 1,&iBytesWritten,NULL);
}

```

7.6. Solving the Flickering Problem

When updating the screen, the screen must not flicker. This problem was solved by drawing the waveform into a memory CDC (bitmap) and then updating the display by drawing only the changes.

Code for updating the display (grid and trace already drawn in their CDC): -

```

void CScopeView::OnPaint()
{
    CPaintDC dc(this); // device context for painting
    CDC memDC;
    CBitmap memBitmap;
    CBitmap* oldBitmap; // bitmap originally found in CMemDC

    // no real plotting work is performed here,
    // just putting the existing bitmaps on the client

    // to avoid flicker, establish a memory dc, draw to it
    // and then BitBlt it to the client
    memDC.CreateCompatibleDC(&dc) ;
    memBitmap.CreateCompatibleBitmap(&dc, m_nClientWidth, m_nClientHeight);
    oldBitmap = (CBitmap *)memDC.SelectObject(&memBitmap);

    if (memDC.GetSafeHdc() != NULL)
    {
        // first drop the grid on the memory dc
        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight, &m_dcGrid, 0, 0, SRCCOPY);

        // now add the plot on top as a "pattern" via SRCPAINT.
        // works well with dark background and a light plot

        /* Only paint channels that's are enabled as this will reduce CPU load */
        if (pDoc->m_bCH1_ENABLE == true) // Only draw if CH1 is enabled
        {
            memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                &m_dcCH1, 0, 0, SRCPAINT); //SRCPAINT
        }

        if (pDoc->m_bCH2_ENABLE == true) // Only draw if CH2 is enabled
        {
            memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                &m_dcCH2, 0, 0, SRCPAINT); //SRCPAINT
        }

        if (pDoc->m_bCH3_ENABLE == true) // Only draw if CH3 is enabled
        {
            memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                &m_dcCH3, 0, 0, SRCPAINT); //SRCPAINT
        }

        if (pDoc->m_bCH4_ENABLE == true) // Only draw if CH4 is enabled
        {

```

```

        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcCH4, 0, 0, SRCPAINT); //SRCPAINT
    }

    // finally send the result to the display
    dc.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
             &memDC, 0, 0, SRCCOPY);

}

memDC.SelectObject(oldBitmap);
}

```

This function (created by wizard) has been modified to override standard windows refresh features: -

```

BOOL CScopeView::Create(LPCTSTR lpszClassName, LPCTSTR lpszWindowName, DWORD dwStyle, const RECT&
rect, CWnd* pParentWnd, UINT nID, CCreateContext* pContext)
{
    /* Overrides the Normal Refresh, stops flicker */
    static CString className = AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW);

    return CWnd::Create(className, lpszWindowName, dwStyle, rect, pParentWnd, nID, pContext);
}

```

7.7. Creating the Split Window View

This function creates the split view, it is called automatically by the framework: -

```

BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)
{
    // create a splitter with 1 row, 2 columns
    if (!m_wndSplitter.CreateStatic(this, 1, 2))
    {
        TRACE0("Failed to CreateStaticSplitter\n");

        return FALSE;
    }

    // add the first splitter pane - the default view in column 0
    if (!m_wndSplitter.CreateView(0, 0,
        pContext->m_pNewViewClass, CSize(350, 100), pContext))
    {
        TRACE0("Failed to create first pane\n");

        return FALSE;
    }

    // add the second splitter pane - an input view in column 1
    if (!m_wndSplitter.CreateView(0, 1,
        RUNTIME_CLASS(CControlsView), CSize(0, 0), pContext))
    {
        TRACE0("Failed to create second pane\n");

        return FALSE;
    }

    // activate the input view
    SetActiveView((CView*)m_wndSplitter.GetPane(0,1));

    m_wndSplitter.LockBar(TRUE);

    m_wndSplitter.SetColumnInfo(1,250,100);

    m_bSplitterCreated = TRUE;
    m_bShowPanel = true;
    return TRUE;
}

```

This function is called when the display is resized, it sets the ratio of the split: -

```

void CChildFrame::OnSize(UINT nType, int cx, int cy)
{
    CMDIChildWnd::OnSize(nType, cx, cy);
    CRect rect;
    GetWindowRect(&rect);
}

```

```
if( m_bSplitterCreated) // m_bSplitterCreated set in OnCreateClient
{
    if (m_bShowPanel == true)
    {
        if(rect.Width()> 250)
        {
            m_wndSplitter.SetColumnInfo(0, rect.Width()-250, 0);
        }
        else
        {
            m_wndSplitter.SetColumnInfo(0,0, 0);
        }

        m_wndSplitter.SetColumnInfo(1, 250, 0);
    }
    else
    {
        m_wndSplitter.SetColumnInfo(0, rect.Width(), 0);
        m_wndSplitter.SetColumnInfo(1, 0, 0);
    }

    m_wndSplitter.RecalcLayout();
}
}
```

This function hides and shows the side panel: -

```
void CChildFrame::HideShowPanel()
{
    CRect rect;
    GetWindowRect( &rect );

    if( m_bSplitterCreated )
    {
        if (m_bShowPanel == true) // Hide side panel
        {
            m_wndSplitter.SetColumnInfo(0, rect.Width(), 0);
            m_wndSplitter.SetColumnInfo(1, 0, 0);
            m_bShowPanel = false;
        }
        else // Show Side Panel
        {
            if(rect.Width()> 200)
            {
                m_wndSplitter.SetColumnInfo(0, rect.Width()-250, 0);
            }
            else
            {
                m_wndSplitter.SetColumnInfo(0,0, 0);
            }
            m_wndSplitter.SetColumnInfo(1, 250, 0);
            m_bShowPanel = true;
        }

        m_wndSplitter.RecalcLayout();
    }
}
```

7.8. Saving / Opening Scope Data

The scope program has the capability of saving and opening waveform data along with the current settings (e.g. selected time-base, sample rate, enabled channels, scope resolution, etc...). This is one of the key advantages that the PC based scope has over traditional analogue scopes. Traditionally users of analogue scopes had to draw on graph paper the scope waveforms if a permanent record was required.

This saving of scope data feature is far more useful than taking a screen dump of the screen or copying the scope waveform to the windows clipboard. Obviously taking a screen dump of the screen, using the windows clipboard, and exporting the scope display to a bitmap are useful for importing an image of the scope display into applications like Microsoft Word, but these images cannot be manipulated (e.g. channel time-base). The saving and opening of the scope data is far more useful for keeping a record of an event for analysis at a later time, because the scope controls are fully functional, for example time-base, center points of waveforms, channel offset, grid resolution and enable/disable channels are adjustable.

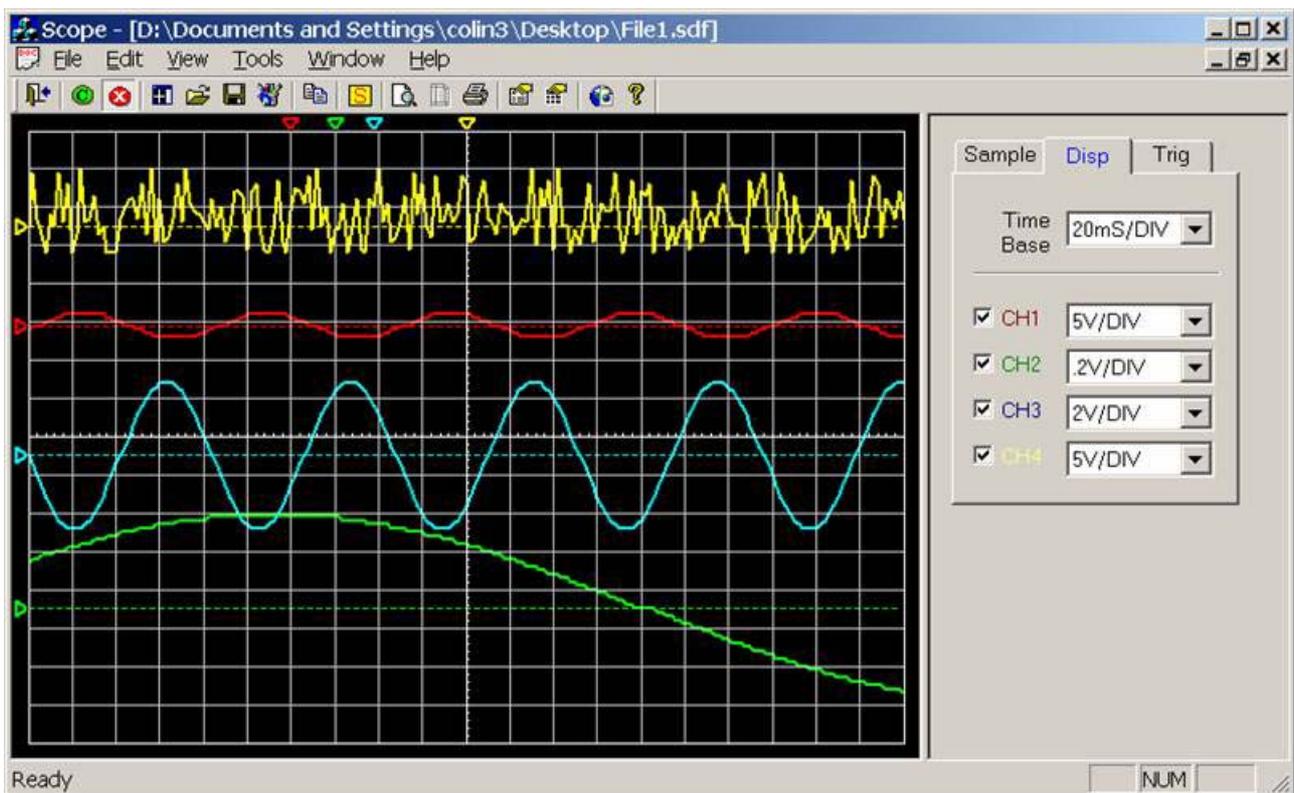


Figure 7.8a. Scope data saved to File1.sdf (Scope V1.040a, 07/04/2002)

*.sdf is the file extension used, it is short for Scope Data File.

If the user clicks on menu item [File] [Save] the following function is called: -

```
void CScopeDoc::OnFileSave ()
{
    // 07/04/2002 by CKM

    if ((_access(m_Filename, 0)) == -1)
    {
        OnFileSaveAs ();
    }
    else
    {
        Save ();
    }
}
```

Basically the OnFileSave() function calls the OnFileSaveAs() function if there is no current file (untitled) or the Save() function to save changes to the current file.

If user clicks on menu item [File] [Save As] the following function is called: -

```
void CScopeDoc::OnFileSaveAs()
{
    // 07/04/2002 by CKM
    CFileDialog dlg (FALSE, _T("*.sdf"), m_Filename, OFN_HIDEREADONLY, _T("Scope Data
    File (*.sdf)|*.sdf|"));

    if (IDOK == dlg.DoModal())
    {
        /* Check for existence */
        if( (_access(dlg.GetPathName(), 0) != -1) )
        {
            if(AfxMessageBox("File '" + dlg.GetPathName() + "' Already exists. Do you
            want to replace the existing file ?", MB_YESNO | MB_ICONINFORMATION) !=
            IDYES)
                return;
        }

        m_Filename = dlg.GetPathName();
        Save();
    }
}
```

Basically the OnFileSaveAs() function using a windows standard CFileDialog box to ask the user to specify a filename and directory. The function also checks if the selected filename already exists, if it does the user gets the option to override the file or to cancel save operation. Function Save() is called, if a valid filename has been entered and user has not cancelled the save operation.

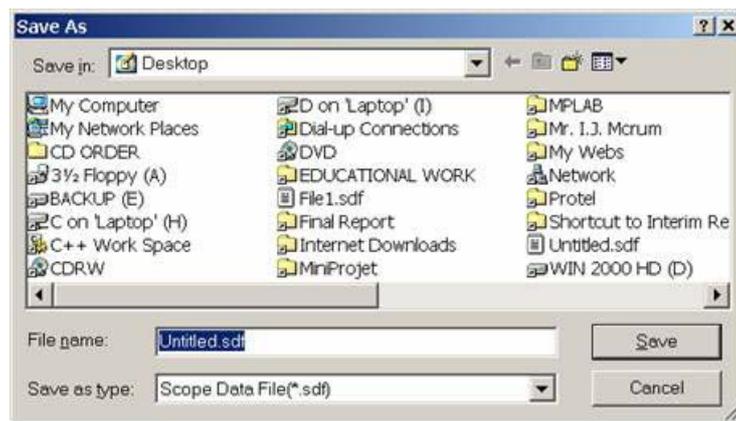


Figure 7.8b. Windows standard CFileDialog box

This function creates and writes the scope data file: -

```
void CScopeDoc::Save()
{
    // 07/04/2002 by CKM
    CView* pView = GetActiveView();
    if (pView == NULL) return;

    CChildFrame* pChild = (CChildFrame*)pView->GetParentFrame();
    CScopeView* SView = (CScopeView*) pChild->m_wndSplitter.GetPane(0,0);

    CScopeApp* pApp = (CScopeApp*)AfxGetApp();
    FILE *infile;

    int i = 0;

    if((infile=fopen(m_Filename,"wr"))==NULL)
    {
        AfxMessageBox("Unable to create file");
        return;
    }

    rewind(infile);

    /* Save Channel 1,2,3 & 4 */
}
```

```

for(i=0;i<10000;i++) fprintf(infile,"%d\n",pApp->m_nCH1Volt[i]);
for(i=0;i<10000;i++) fprintf(infile,"%d\n",pApp->m_nCH2Volt[i]);
for(i=0;i<10000;i++) fprintf(infile,"%d\n",pApp->m_nCH3Volt[i]);
for(i=0;i<10000;i++) fprintf(infile,"%d\n",pApp->m_nCH4Volt[i]);

/* Save Variables */
fprintf(infile,"%d\n", SView->x);
fprintf(infile,"%d\n", SView->y);
fprintf(infile,"%d\n",SView->m_nCH1POS);
fprintf(infile,"%d\n",SView->m_nCH2POS);
fprintf(infile,"%d\n",SView->m_nCH3POS);
fprintf(infile,"%d\n",SView->m_nCH4POS);
fprintf(infile,"%d\n",SView->m_nCH1_Offset);
fprintf(infile,"%d\n",SView->m_nCH2_Offset);
fprintf(infile,"%d\n",SView->m_nCH3_Offset);
fprintf(infile,"%d\n",SView->m_nCH4_Offset);
fprintf(infile,"%d\n",pApp->m_nInputRange);
fprintf(infile,"%d\n",pApp->m_nSampleMode);
fprintf(infile,"%d\n",pApp->m_SampleRateSelection);
fprintf(infile,"%d\n",m_bCH1_ENABLE);
fprintf(infile,"%d\n",m_bCH2_ENABLE);
fprintf(infile,"%d\n",m_bCH3_ENABLE);
fprintf(infile,"%d\n",m_bCH4_ENABLE);
fprintf(infile,"%d\n",m_TimeBaseSelection);
fprintf(infile,"%d\n",m_nCH1_VOLTS);
fprintf(infile,"%d\n",m_nCH2_VOLTS);
fprintf(infile,"%d\n",m_nCH3_VOLTS);
fprintf(infile,"%d\n",m_nCH4_VOLTS);
fprintf(infile,"%d\n",m_nTrigger);
fprintf(infile,"%d\n",m_nTriggerType);
fprintf(infile,"%d\n",SView->m_ColorGridBackGround);
fprintf(infile,"%d\n",SView->m_nRefreshRate);

fclose(infile);

SetTitle(m_Filename);
}

```

Notice that function Save() uses the old ASCII C method for saving files, this method is 100% windows compatible (including long filenames) and is preferred by most C++ programmers. Basically a pointer to the file is setup (*infile) and the ASCII C function fprintf() is used to save scope data to the file using the ASCII format. Writing the file in binary and not ASCII would produce a more efficient file, but this file would not be readable in notepad for example. First the four data arrays (one for each channel) are saved, then all the key variables are saved (e.g. channel center point position m_nCH1POS, channel offset m_nCH1_Offset etc...).

If user clicks on menu item [File] [Open] the following function is called: -

```

void CScopeDoc::OnFileOpen ()
{
    // 07/04/2002 by CKM
    CView* pView = GetActiveView();
    if (pView == NULL) return;

    CChildFrame* pChild = (CChildFrame*)pView->GetParentFrame();
    CScopeView* SView = (CScopeView*) pChild->m_wndSplitter.GetPane(0,0);

    CScopeApp* pApp = (CScopeApp*)AfxGetApp();
    FILE *infile;
    int i;
    int temp;

    CFileDialog dlg (TRUE, _T("*.sdf"), m_Filename, OFN_HIDEREADONLY, _T("Scope Data
    File (*.sdf) | *.sdf|"));

    if (IDOK == dlg.DoModal())
    {
        m_Filename = dlg.GetPathName();

        if ((infile=fopen(m_Filename,"rw"))==NULL)
        {
            AfxMessageBox("Unable to open file");
            return;
        }
    }
}

```

```

rewind(infile);

/* Open Channel 1 */
for(i=0;i<10000;i++)
{
    fscanf(infile,"%d\n",&temp);
    pApp->m_nCH1Volt[i] = temp;
}

/* Open Channel 2 */
for(i=0;i<10000;i++)
{
    fscanf(infile,"%d\n",&temp);
    pApp->m_nCH2Volt[i] = temp;
}

/* Open Channel 3 */
for(i=0;i<10000;i++)
{
    fscanf(infile,"%d\n",&temp);
    pApp->m_nCH3Volt[i] = temp;
}

/* Open Channel 4 */
for(i=0;i<10000;i++)
{
    fscanf(infile,"%d\n",&temp);
    pApp->m_nCH4Volt[i] = temp;
}

fscanf(infile,"%d\n",&SView->x);
fscanf(infile,"%d\n",&SView->y);
fscanf(infile,"%d\n",&SView->m_nCH1POS);
fscanf(infile,"%d\n",&SView->m_nCH2POS);
fscanf(infile,"%d\n",&SView->m_nCH3POS);
fscanf(infile,"%d\n",&SView->m_nCH4POS);
fscanf(infile,"%d\n",&SView->m_nCH1_Offset);
fscanf(infile,"%d\n",&SView->m_nCH2_Offset);
fscanf(infile,"%d\n",&SView->m_nCH3_Offset);
fscanf(infile,"%d\n",&SView->m_nCH4_Offset);
fscanf(infile,"%d\n",&pApp->m_nInputRange);
fscanf(infile,"%d\n",&pApp->m_nSampleMode);
fscanf(infile,"%d\n",&pApp->m_nSampleRateSelection);
fscanf(infile,"%d\n",&m_bCH1_ENABLE);
fscanf(infile,"%d\n",&m_bCH2_ENABLE);
fscanf(infile,"%d\n",&m_bCH3_ENABLE);
fscanf(infile,"%d\n",&m_bCH4_ENABLE);
fscanf(infile,"%d\n",&m_nTimeBaseSelection);
fscanf(infile,"%d\n",&m_nCH1_VOLTS);
fscanf(infile,"%d\n",&m_nCH2_VOLTS);
fscanf(infile,"%d\n",&m_nCH3_VOLTS);
fscanf(infile,"%d\n",&m_nCH4_VOLTS);
fscanf(infile,"%d\n",&m_nTrigger);
fscanf(infile,"%d\n",&m_nTriggerType);
fscanf(infile,"%d\n",&SView->m_nColorGridBackGround);
fscanf(infile,"%d\n",&SView->m_nRefreshRate);

fclose(infile);

UpdateAllViews(NULL,HINT_OPEN,NULL);
SView->PrepareGrid();
SView->PrepareTraces();
SView->OnDrawGrid();
SView->OnDrawCH1();
SView->OnDrawCH2();
SView->OnDrawCH3();
SView->OnDrawCH4();
}
SetTitle(m_Filename);
}

```

A standard Windows CFileDialog is used to allow the user to select the file to open, then a pointer to the file is setup. Once again the old ASIC C method of accessing files is used, this time using the function fscanf() to read the file line-by-line. The data is retrieved from the data file in exactly the same order as it was saved; once all fields have been acquired the scope display is refreshed.

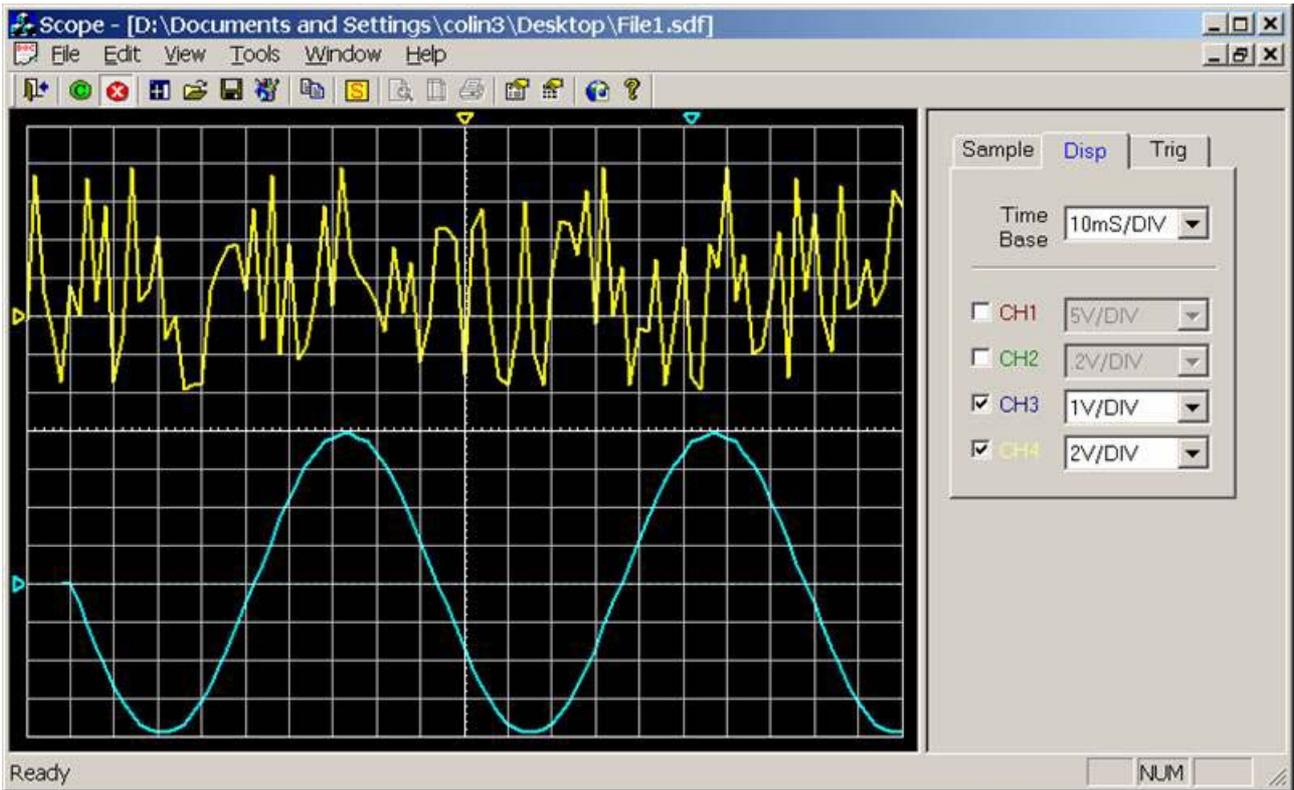


Figure 7.8c. Example: File1.sdf has been opened, then CH1/CH2 was disabled with voltage / time-base scales modified

Figure 7.8c. demonstrates the potential of this feature, File1.sdf has been loaded. Channel 1 and Channel 2 has been disabled (Hidden), voltage and time-base scales have been modified. Zero point of channels 3 and 4 has been moved and a large offset has been applied to channel 3.

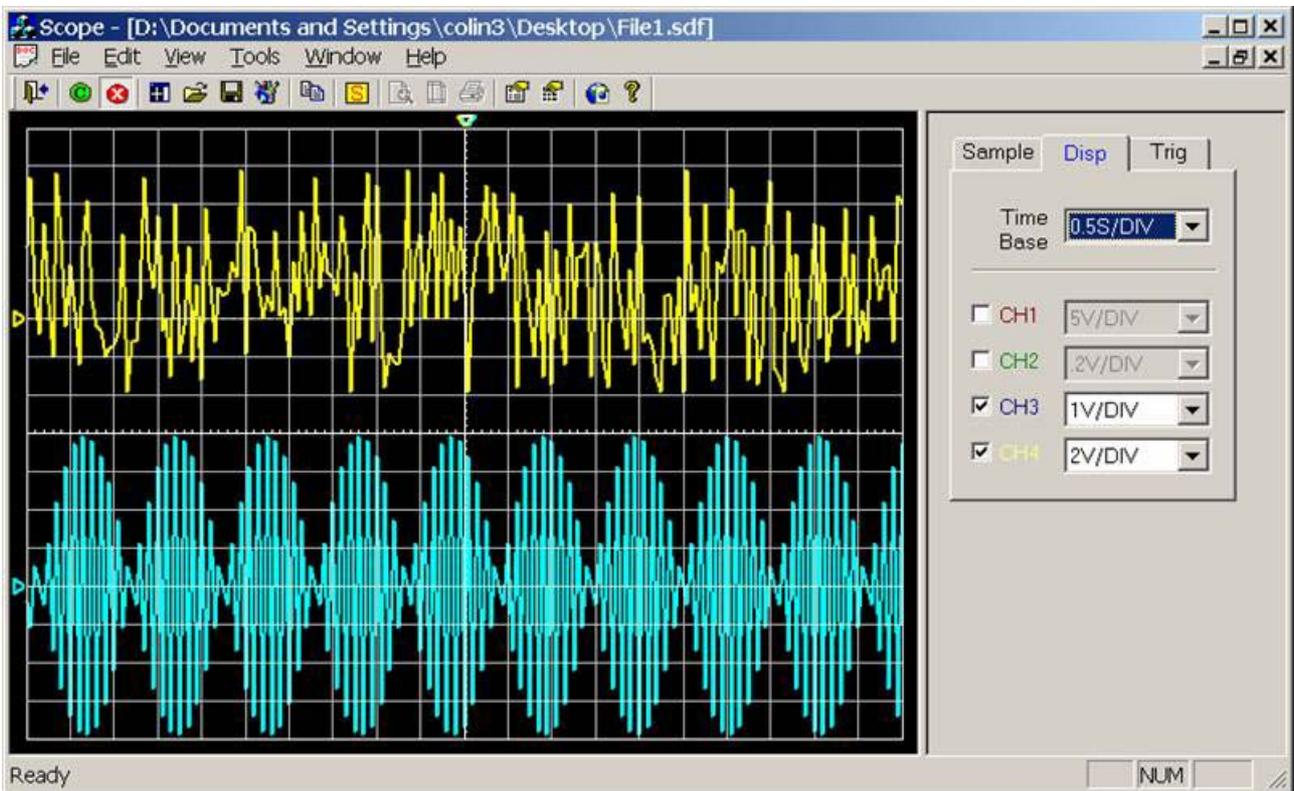


Figure 7.8d. Demonstrating aliasing by selecting a out of range time-base

It is important to remember that the saved waveforms are with respect to the original sample rate. The scope program increases the time-base by skipping readings and decreases the time-base by duplicating readings. As shown in figure 7.8d. it is possible for aliasing to occur if the selected time-base requires for less than 2 samples per cycle to be displayed, hence it is important to make sure the sample-rate is optimal for the waveform being monitored before saving.

File1.sdf (235kB): -

```
0
298
589
867
1126
1360
1563
1732
1861
1949
1994
1994
1949
1861
1732
1563
1360
1126
867
589
298
0
-298
-589
-867
-1126
-1360
-1563
-1732
-1861
-1949
-1994
-1994
-1949
-1861
-1732
-1563
-1360
-1126
-867
-589
-298
0
298
589
867
1126
1360
1563
1732
1861
1949
1994
1994
1949
1861
1732
1563
1360
1126
```

The file format is simple the data is listed in one large column. The data shown above is part of the data for channel 1; clearly it is a sine-wave. The values are actual voltage readings that have been multiplied by 1000 to improve accuracy, dividing by 1000 will convert back to the voltage, for example 298 is 0.298 V, 589 is 5.89V and the peak 1994 is 1.994 Volts.

7.9. Copying the Scope Display to the Windows Clipboard

The scope program has the ability to copy the scope display to the windows clipboard in the form of a bitmap; the paste command in many Windows applications (e.g. Word) will import the scope image. The user can use the right-click menu (as shown in figure 7.9a), the edit menu, the shortcut key [Ctrl+C], or the copy toolbar icon.

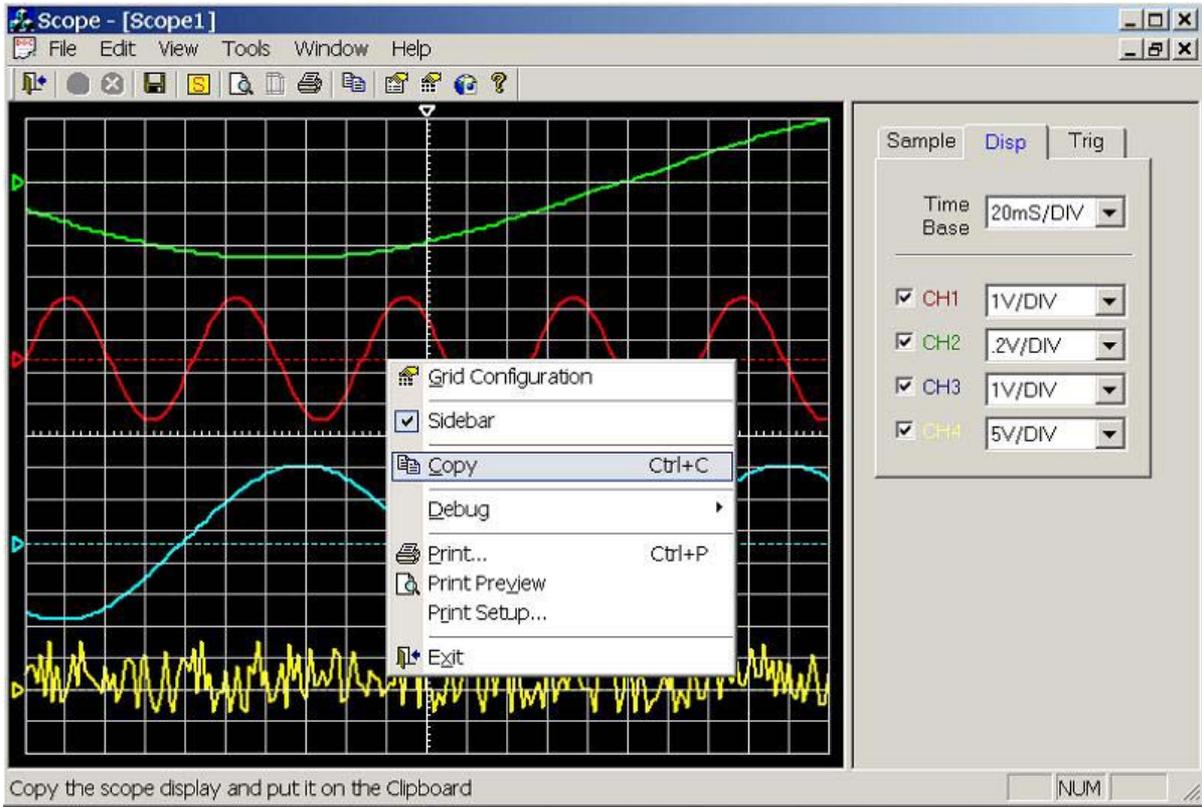


Figure 7.9a. Screen dump demonstrating how to copy the scope display to the clipboard (Scope V1.038a, 04/04/2002)

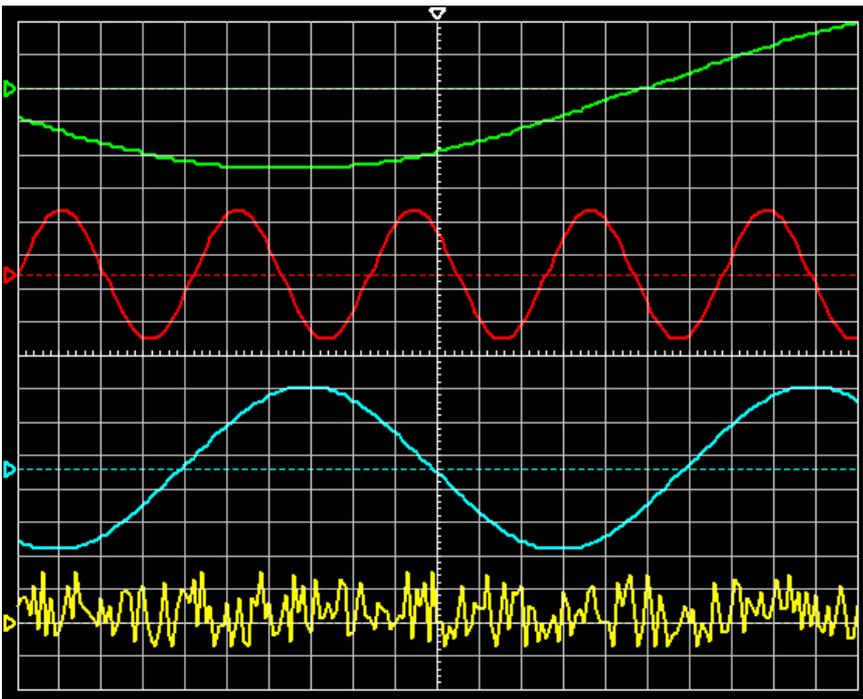


Figure 7.9b. Scope display was pasted into this document from the clipboard

Figure 7.9b is an image of the scope display that was copied into the Windows clipboard and pasted into Microsoft Word (This document).

Only the scope waveforms are copied and not the entire scope display as was the case for the screen dump shown in figure 7.9a.

It maybe useful for other information like time-base and channel voltage scales to be included. This change is recommended and should be adopted sometime in the future, because the scope waveforms are useless without knowledge of the time-base and voltages scales used.

This function copies the scope display to the clipboard: -

```

void CScopeView::OnEditCopy()
{
/* -----
| Fuction:      void CScopeView::OnEditCopy()      |
| Description:  Copies the scope display to the Windows |
|              clipboard.                          |
| Return:      Void.                                |
| Date:        15/02/2002                          |
| Verison:     1.0                                  |
| By:         Colin McCord                          |
|-----*/

    CRect      rect;
    CClientDC  dc(this);
    CDC        memDC;
    CBitmap    bitmap;

    GetClientRect(&rect);

    // Create memDC
    memDC.CreateCompatibleDC(&dc);
    bitmap.CreateCompatibleBitmap(&dc, rect.Width(), rect.Height());
    CBitmap* pOldBitmap = memDC.SelectObject(&bitmap);

    // Fill in memDC
    memDC.FillSolidRect(rect, dc.GetBkColor());

    // Redraw grid to memory DC
    if (memDC.GetSafeHdc() != NULL)
    {
        // first drop the grid on the memory dc
        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcGrid, 0, 0, SRCCOPY);

        // now add the plot on top as a "pattern" via SRCPAINT.
        // works well with dark background and a light plot
        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcCH1, 0, 0, SRCPAINT); //SRCPAINT

        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcCH2, 0, 0, SRCPAINT); //SRCPAINT

        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcCH3, 0, 0, SRCPAINT); //SRCPAINT

        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcCH4, 0, 0, SRCPAINT); //SRCPAINT
    }

    // Copy contents of memDC to clipboard
    OpenClipboard();
    EmptyClipboard();
    SetClipboardData(CF_BITMAP, bitmap.GetSafeHandle());
    CloseClipboard();

    // Clean up
    memDC.SelectObject(pOldBitmap);
    bitmap.Detach();
}

```

7.10. Exporting the Scope Display as a Bitmap (BMP) File

The scope program has the ability to save the scope display as a bitmap file. The user clicks on [Export] in the [File] menu (see figure 7.10a), then a standard windows save dialogue box appears (see figure 7.10b) to ask the user where to save the file.

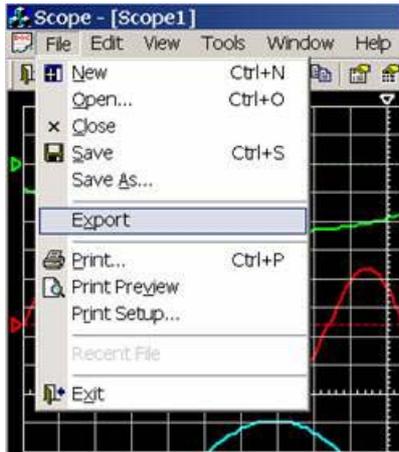


Figure 7.10a. User clicks [Export]

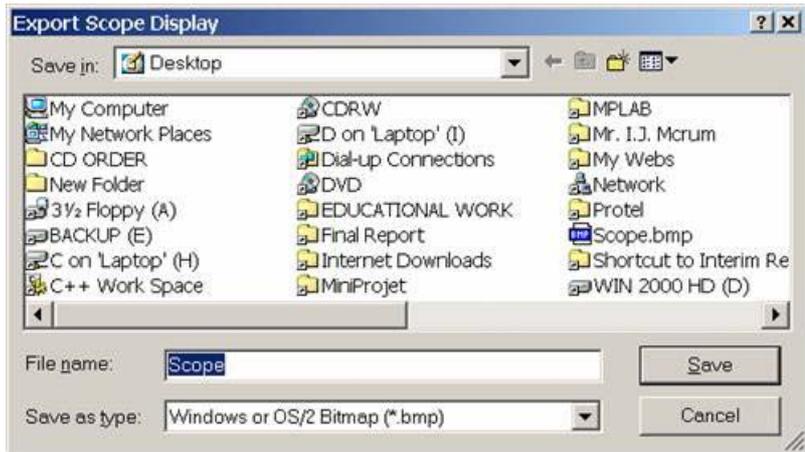


Figure 7.10b. Save dialogue box appears

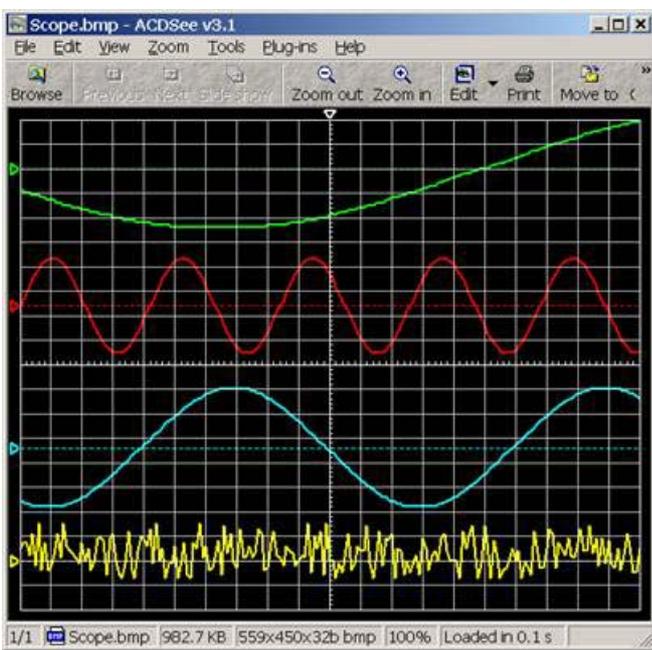


Figure 7.10c. Screen dump of exported file scope.bmp

Figure 7.10c shows the contents of the exported file scope.bmp.

Saving the scope display to a BMP file would have been fairly simple if a handle to a device-independent bitmap existed. Simply write BITMAPINFOHEADER information followed by the contents of the bitmap. The three fields that have to be set in the BITMAPINFOHEADER are the bfType which should always be "BM", the bfSize which is the size of the bitmap including the infoheader and the bfOffBits which is the offset to the bitmap bits from the start of the file.

But unfortunately the scope display is a device-dependent bitmap. Therefore a DIB (Device Independent Bitmap) must first be created from it.

The following function converts a DDB (Device Dependent Bitmap) to a DIB (Device Independent Bitmap): -

```
HANDLE CScopeView::DDBToDIB(CBitmap &bitmap, DWORD dwCompression, CPalette *pPal)
{
/*
-----
| Function:      HANDLE CScopeView::DDBToDIB(...)
| Description:   Convert Device-Dependent Bitmap (DDB) to
|               Device-independent Bitmap (DIB).
| Return:       Void.
| Date:         16/02/2002
| Verison:      1.3
| By:          Colin K McCord
-----
*/

// DDBToDIB          - Creates a DIB from a DDB
// bitmap            - Device dependent bitmap
// dwCompression-    Type of compression - see BITMAPINFOHEADER
// pPal              - Logical palette

BITMAP                bm;
BITMAPINFOHEADER      bi;
LPBITMAPINFOHEADER    lpbi;
DWORD                 dwLen;
HANDLE                 hDIB;
HANDLE                 handle;
HDC                   hDC;
```

```

HPALETTE                hPal;

ASSERT( bitmap.GetSafeHandle() );

// The function has no arg for bitfields
if( dwCompression == BI_BITFIELDS )
    return NULL;

// If a palette has not been supplied use default palette
hPal = (HPALETTE) pPal->GetSafeHandle();
if (hPal==NULL)
    hPal = (HPALETTE) GetStockObject(DEFAULT_PALETTE);

// Get bitmap information
bitmap.GetObject( sizeof(bm), (LPSTR) &bm);

// Initialize the bitmapinfoheader
bi.biSize                = sizeof(BITMAPINFOHEADER);
bi.biWidth               = bm.bmWidth;
bi.biHeight              = bm.bmHeight;
bi.biPlanes              = 1;
bi.biBitCount            = bm.bmPlanes * bm.bmBitsPixel;
bi.biCompression        = dwCompression;
bi.biSizeImage           = 0;
bi.biXPelsPerMeter       = 0;
bi.biYPelsPerMeter       = 0;
bi.biClrUsed             = 0;
bi.biClrImportant        = 0;

// Compute the size of the infoheader and the color table
int nColors = (1 << bi.biBitCount);

dwLen = bi.biSize + nColors * sizeof( RGBQUAD );

// We need a device context to get the DIB from
hDC = ::GetDC( NULL );

hPal = SelectPalette( hDC, hPal, FALSE );
RealizePalette( hDC );

// Allocate enough memory to hold bitmapinfoheader and color table
hDIB = GlobalAlloc( GMEM_FIXED, dwLen );

if ( !hDIB )
{
    SelectPalette( hDC, hPal, FALSE );
    ::ReleaseDC( NULL, hDC );
    return NULL;
}

lpbi = (LPBITMAPINFOHEADER) hDIB;

*lpbi = bi;

// Call GetDIBits with a NULL lpBits param, so the device driver
// will calculate the biSizeImage field
GetDIBits( hDC, (HBITMAP) bitmap.GetSafeHandle(), 0L, (DWORD) bi.biHeight,
           (LPBYTE) NULL, (LPBITMAPINFO) lpbi, (DWORD) DIB_RGB_COLORS );

bi = *lpbi;

// If the driver did not fill in the biSizeImage field, then compute it
// Each scan line of the image is aligned on a DWORD (32bit) boundary
if ( bi.biSizeImage == 0 ) {
    bi.biSizeImage = (((bi.biWidth * bi.biBitCount) + 31) & ~31) / 8)
                    * bi.biHeight;

    // If a compression scheme is used the result may infact be larger
    // Increase the size to account for this.
    if ( dwCompression != BI_RGB )
        bi.biSizeImage = (bi.biSizeImage * 3) / 2;
}

// Realloc the buffer so that it can hold all the bits
dwLen += bi.biSizeImage;
if ( handle = GlobalReAlloc( hDIB, dwLen, GMEM_MOVEABLE ) )

```

```

        hDIB = handle;
    else{
        GlobalFree(hDIB);

        // Reselect the original palette
        SelectPalette(hDC,hPal,FALSE);
        ::ReleaseDC(NULL,hDC);
        return NULL;
    }

    // Get the bitmap bits
    lpbi = (LPBITMAPINFOHEADER)hDIB;

    // FINALLY get the DIB
    BOOL bGotBits = GetDIBits( hDC, (HBITMAP)bitmap.GetSafeHandle(),
                               0L, // Start scan line
                               (DWORD)bi.biHeight, // # of scan lines
                               (LPBYTE)lpbi // address for bitmap bits
                               + (bi.biSize + nColors * sizeof( RGBQUAD)),
                               (LPBITMAPINFO)lpbi, // address of bitmapinfo
                               (DWORD)DIB_RGB_COLORS); // Use RGB for colour table

    if( !bGotBits )
    {
        GlobalFree(hDIB);
        SelectPalette(hDC,hPal,FALSE);
        ::ReleaseDC(NULL,hDC);
        return NULL;
    }

    SelectPalette(hDC,hPal,FALSE);
    ::ReleaseDC(NULL,hDC);
    return hDIB;
}

```

Convert the scope display to a bitmap (DDB) and ask the user for a filename: -

```

void CScopeView::OnFileExport()
{
    /* -----
    | Fuction:      void CScopeView::OnFileExport() |
    | Description:  User specifies filename using CFileDialog, |
    |              then the scope display is save to that |
    |              filename using the standard bitmap |
    |              graphic format. |
    | Return:      Void. |
    | Date:        16/02/2002 |
    | Verison:     1.1 |
    | By:          Colin K McCord |
    |----- */

    HANDLE          hDIB;
    CRect           rect;
    CClientDC       dc(this);
    CDC             memDC;
    CBitmap         bitmap;

    GetClientRect(&rect);

    // Create memDC
    memDC.CreateCompatibleDC(&dc);
    bitmap.CreateCompatibleBitmap(&dc, rect.Width(), rect.Height());
    CBitmap* pOldBitmap = memDC.SelectObject(&bitmap);

    // Fill in memDC
    memDC.FillSolidRect(rect, dc.GetBkColor());

    // Redraw grid to memory DC
    if (memDC.GetSafeHdc() != NULL)
    {
        // first drop the grid on the memory dc
        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                   &m_dcGrid, 0, 0, SRCCOPY);
    }
}

```

```

        // now add the plot on top as a "pattern" via SRCPAINT.
        // works well with dark background and a light plot
        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcCH1, 0, 0, SRCPAINT); //SRCPAINT

        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcCH2, 0, 0, SRCPAINT); //SRCPAINT

        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcCH3, 0, 0, SRCPAINT); //SRCPAINT

        memDC.BitBlt(0, 0, m_nClientWidth, m_nClientHeight,
                    &m_dcCH4, 0, 0, SRCPAINT); //SRCPAINT

    }

    /* Convert Device-Dependent Bitmap (DDB) to Device-independent Bitmap (DIB) */
    hDIB = DDBToDIB(bitmap, BI_RGB, NULL);

    CFileDialog dlg (FALSE, _T("*.bmp"), _T("Scope"), OFN_HIDEREADONLY, _T("Windows or OS/2
    Bitmap (*.bmp)|*.bmp|"));
    dlg.m_ofn.lpstrTitle = "Export Scope Display";
    if (IDOK == dlg.DoModal())
    {

        /* Check for existence */
        if( (_access(dlg.GetPathName(), 0) ) != -1 )
        {
            if(MessageBox("File '" + dlg.GetPathName() + "' Already exists. Do you want
            to replace the existing file ?", "Export Scope Display", MB_YESNO |
            MB_ICONINFORMATION) != IDYES)
                return;
        }

        WriteDIB(dlg.GetPathName(), hDIB);    // Save as Bitmap

    }
}

```

Write bitmap to specified file: -

```

BOOL CScopeView::WriteDIB(CString szFile, HANDLE hDIB)
{
    /* -----
    | Fuction:      BOOL CScopeView::WriteDIB(.....)      |
    | Description:  Writes bitmap to specified file.      |
    | Return:      Void.                                  |
    | Date:        16/02/2002                             |
    | Verison:     1.1                                    |
    | By:          Colin K McCord                         |
    |-----*/

    // WriteDIB          - Writes a DIB to file
    // Returns            - TRUE on success
    // szFile             - Name of file to write to
    // hDIB               - Handle of the DIB

    BITMAPFILEHEADER    hdr;
    LPBITMAPINFOHEADER  lpbi;

    if (!hDIB)
        return FALSE;

    CFile file;
    if( !file.Open( szFile, CFile::modeWrite|CFile::modeCreate) )
        return FALSE;

    lpbi = (LPBITMAPINFOHEADER)hDIB;

    int nColors = 1 << lpbi->biBitCount;

    // Fill in the fields of the file header
    hdr.bfType      = ((WORD) ('M' << 8) | 'B'); // is always "BM"
    hdr.bfSize      = GlobalSize (hDIB) + sizeof( hdr );

```

```

hdr.bfReserved1      = 0;
hdr.bfReserved2      = 0;
hdr.bfOffBits        = (DWORD) (sizeof( hdr ) + lpbi->biSize +
                                nColors * sizeof( RGBQUAD ) );

// Write the file header
file.Write( &hdr, sizeof(hdr) );

// Write the DIB header and the bits
file.Write( lpbi, GlobalSize(hDIB) );

return TRUE;
}

```

7.11. Owner Drawn Menus with Bitmaps

The scope program uses owner drawn menus with bitmaps. To save time a freeware class (BCMenu.h) written by Brent Corkum was used (download from [W4]). This class can mimic the new MS Office XP style of menus (Used in scope program), or the old MS Office 97 style of menus. Figures 7.11a shows a screen dump of Microsoft Word XP with its file menu open, while figure 7.11b shows the file menu of the Scope program. Notice that the menu style of the Scope program is similar, but not perfect, to that of Word XP. It is clear that Brent Corkum has done a good job developing this class and it was decided after testing on Windows 98/2000/XP for the scope program to make use of his class as it made the program more professional looking and more user-friendly (Icons in the menus).

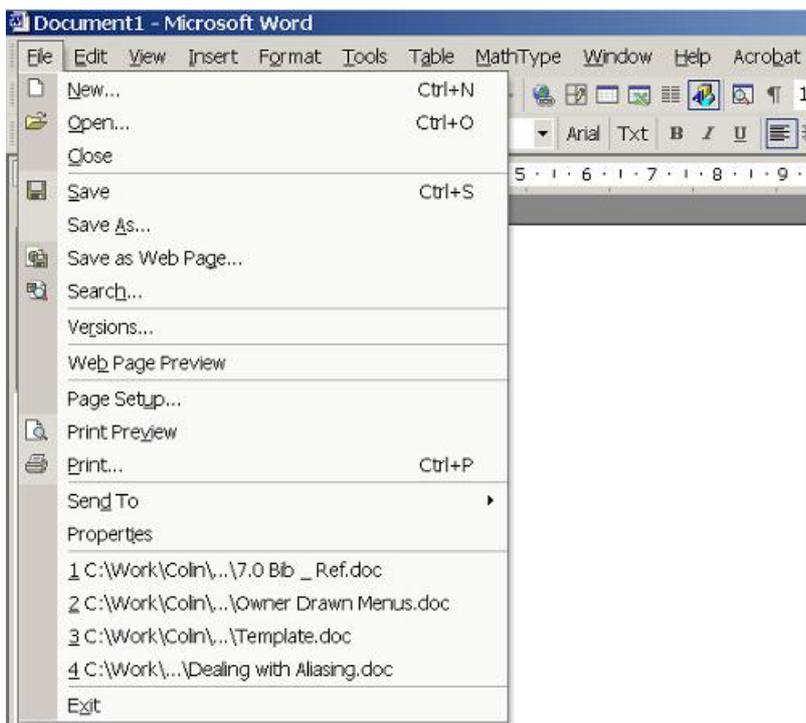


Figure 7.11a. Microsoft Word XP file menu

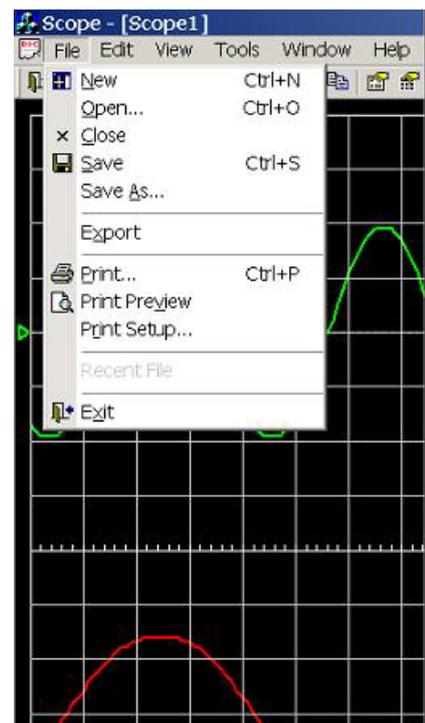


Figure 7.11b. Scope Ver1.036a file Menu

“This class, BCMenu, implements owner drawn menus derived from the CMenu class. The purpose of which is to mimic the menu style used in Visual C++ 5.0 and MS Word. I can't take credit for all the code; some portions of it were taken from code supplied by Ben Ashley and Girish Bharadwaj. The difference between their codes and this one is quite simple; this one makes it very easy to build these cool menus with bitmaps into your application. I've removed the icon loading stuff and replaced it with Bitmaps. The bitmaps allow you to use the 16X15 toolbar bitmaps directly from your toolbars in the resource editor. As well, there is no scaling of the bitmaps so they always look good. You can also load Bitmap resources and define bitmaps for your check marks. I've also added the default checkmark drawing stuff, separators, proper alignment of keyboard accelerator text, keyboard shortcuts, proper alignment of popup menu items, proper system colour changes when the Display Appearance changes, plus bug fixes to the Ben Ashley's LoadMenu function for

complex submenu systems. I made quite a few other modifications as well, too many to list or remember. I also use the disabled bitmap dithering function of Jean-Edouard Lachand-Robert to create the disabled state bitmaps. I must admit, it does a much better job than the DrawState function. If you find any bugs, memory leaks, or just better ways of doing things, please let me know. I used Visual C++ 5.0 and I have not tested compatibility with earlier VC versions. I've tested it on Win 95/NT at various resolutions and colour palette sizes." Brent Corkum [W4]

Title block from BCMenu.h: -

```

//*****
// BCMenu.h : header file
// Version : 3.0
// Date : January 2002
// Author : Brent Corkum
// Email : corkum@rocscience.com
// Latest Version : http://www.rocscience.com/~corkum/BCMenu.html
//
// Bug Fixes and portions of code supplied by:
//
// Ben Ashley, Girish Bharadwaj, Jean-Edouard Lachand-Robert,
// Robert Edward Caldecott, Kenny Goers, Leonardo Zide,
// Stefan Kuhr, Reiner Jung, Martin Vladic, Kim Yoo Chul,
// Oz Solomonovich, Tongzhe Cui, Stephane Clog, Warren Stevens,
// Damir Valiulin
//
// You are free to use/modify this code but leave this header intact.
// This class is public domain so you are free to use it any of
// your applications (Freeware, Shareware, Commercial). All I ask is
// that you let me know so that if you have a real winner I can
// brag to my buddies that some of my code is in your app. I also
// wouldn't mind if you sent me a copy of your application since I
// like to play with new stuff.
//*****

```

7.11.1. Integrating Brent Corkum's BCMenu Class with the Scope Program

Step 1: Add #include "BCMenu.cpp" to stdafx.cpp: -

```

// stdafx.cpp : source file that includes just the standard includes
// Scope.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information
#include "stdafx.h"
#include "UsefulSplitterWnd.cpp"
#include "XTabCtrl.cpp"
#include "StatLink.cpp"
#include "BCMenu.cpp"
#include "ColorStatic.cpp"

```

Step 2: Add #include "BCMenu.h", m_default and m_menu to MainFrm.h: -

```

// MainFrm.h : interface of the CMainFrame class
//
////////////////////////////////////////////////////////////////////
#if !defined(AFX_MAINFRM_H__8C66EB30_6DEB_4044_863D_D90EA5FE5723__INCLUDED_)
#define AFX_MAINFRM_H__8C66EB30_6DEB_4044_863D_D90EA5FE5723__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "BCMenu.h" // Bitmap Menus

class CMainFrame : public CMDIFrameWnd
...
public:
    void ComRealTimeBuffered();
    void ComRealTimeScroll();
    BCMenu m_default;
    BCMenu m_menu;
...

```

Step 3: Add function NewMenu to class CMainFrame (Notice menu bitmaps are from toolbars): -

```

HMENU CMainFrame::NewMenu()
{
    static UINT toolbars[]=
    {
        IDR_MAINFRAME,
        IDR_TOOLBAR
    };

    // Load the menu from the resources
    m_menu.LoadMenu(IDR_SCOPETYPE);

    // One method for adding bitmaps to menu options is
    // through the LoadToolbars member function. This method
    // allows you to add all the bitmaps in a toolbar object
    // to menu options (if they exist). The first function
    // parameter is an array of toolbar id's. The second is
    // the number of toolbar id's. There is also a function
    // called LoadToolbar that just takes an id.
    m_menu.LoadToolbars(toolbars,2);

    return(m_menu.Detach());
}

```

Step 4: Add function NewDefaultMenu() to class CMainFrame: -

```

HMENU CMainFrame::NewDefaultMenu()
{
    m_default.LoadMenu(IDR_MAINFRAME);
    m_default.LoadToolBar(IDR_MAINFRAME);
    return(m_default.Detach());
}

```

Step 5: Edit InitInstance() function in class CScopeApp: -

```

BOOL CScopeApp::InitInstance()
{
    ...

    // create main MDI Frame window
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
    m_pMainWnd = pMainFrame;

    // This code replaces the MFC created menus with
    // Ownerdrawn versions
    pDocTemplate->m_hMenuShared=pMainFrame->NewMenu();
    pMainFrame->m_hMenuDefault=pMainFrame->NewDefaultMenu();

    // This simulates a window being opened if you don't have
    // a default window displayed at start-up
    pMainFrame->OnUpdateFrameMenu(pMainFrame->m_hMenuDefault);

    ...
}

```

Step 6: Add the message handler for the WM_MEASUREITEM message: -

```

void CMainFrame::OnMeasureItem(int nIDCtl, LPMEASUREITEMSTRUCT lpMeasureItemStruct)
{
    //This handler ensure that the popup menu items are
    // drawn correctly

    BOOL setflag=FALSE;

    if(lpMeasureItemStruct->CtlType==ODT_MENU)
    {
        if(IsMenu((HMENU)lpMeasureItemStruct->itemID))
        {
            CMenu* cmenu =
            CMenu::FromHandle((HMENU)lpMeasureItemStruct->itemID);

            if(m_menu.IsMenu(cmenu) || m_default.IsMenu(cmenu))
            {
                m_menu.MeasureItem(lpMeasureItemStruct);
                setflag=TRUE;
            }
        }
    }
}

```

```

    }
}

if (!setflag) CMDIFrameWnd::OnMeasureItem(nIDCtl, lpMeasureItemStruct);
}

```

Step 7: Add the message handler for the WM_MENUCHAR message: -

```

LRESULT CMainFrame::OnMenuChar(UINT nChar, UINT nFlags, CMenu* pMenu)
{
    //This handler ensures that keyboard shortcuts work
    LRESULT lresult;
    if (m_menu.IsMenu(pMenu) || m_default.IsMenu(pMenu))
        lresult = BCMenu::FindKeyboardShortcut(nChar, nFlags, pMenu);
    else
        lresult = CMDIFrameWnd::OnMenuChar(nChar, nFlags, pMenu);
    return (lresult);
}

```

Step 8: Add the message handler for the WM_INITMENUPOPUP message: -

```

void CMainFrame::OnInitMenuPopup(CMenu* pPopupMenu, UINT nIndex, BOOL bSysMenu)
{
    //This handler updates the menus from time to time
    CMDIFrameWnd::OnInitMenuPopup(pPopupMenu, nIndex, bSysMenu);

    CMDIFrameWnd::OnInitMenuPopup(pPopupMenu, nIndex, bSysMenu);
    if (!bSysMenu)
    {
        if (m_menu.IsMenu(pPopupMenu) || m_default.IsMenu(pPopupMenu))
            BCMenu::UpdateMenu(pPopupMenu);
    }
}

```

Step 9: Specify XP menu style for all versions of Microsoft windows and not just XP (Edit BCMenu.cpp): -

```

#include "stdafx.h" // Standard windows header file
#include "BCMenu.h" // BCMenu class declaration
#include <afxpriv.h> // SK: makes A2W and other spiffy AFX macros work

...

static CPINFO CPInfo;
// how the menu's are drawn in win9x/NT/2000
UINT BCMenu::original_drawmode=BCMENU_DRAWMODE_XP; //Use BCMENU_DRAWMODE_ORIGINAL for Original
BOOL BCMenu::xp_select_disabled=FALSE;
// how the menu's are drawn in winXP
UINT BCMenu::xp_drawmode=BCMENU_DRAWMODE_XP;
BOOL BCMenu::original_select_disabled=TRUE;

enum Win32Type{
    Win32s,
    Windoze95,
    WinNT3,
    WinNT4orHigher
};

...

```

That's it; the scope program now has professional looking owner drawn MS Office XP style bitmap menus.

Step 10: This class is also applied to the right click pop-up menu: -

```

void CMainFrame::OnContextMenu(CWnd* pWnd, CPoint point)
{
    BCMenu popmenu;

    popmenu.LoadMenu(IDR_GRID_POPUP);
    popmenu.LoadToolBar(IDR_MAINFRAME);

    BCMenu *psub = (BCMenu *)popmenu.GetSubMenu(0);
    psub->TrackPopupMenu(TPM_LEFTALIGN|TPM_RIGHTBUTTON, point.x, point.y, this);
    popmenu.DestroyMenu();
}

```

7.12. Windows 95/98 Compatibility Problem (scope v1.008a, 20/12/2001)

The 'Scope' program is being developed on Windows 2000 (service pack 2); Windows 2000 is a stable and reliable operating system, unlike other Microsoft operating systems (e.g. Windows 95, Windows 98 and Windows ME). It is important that the scope program runs on all versions of Microsoft Windows, as Windows 95, 98 and ME are still widely used.

On the 20th of December 2001, the scope program (Version 1.008a) was tested on Windows 98 and it was discovered there was a significant problem. The CPU load was high, and the system memory resources were disappearing extremely quickly, after about a minute the system was completely unusable. At first it was not understood why this was happening, as the program worked on Windows 2000 without any problems, with a CPU load of less than 2%. The tests were carried out on the same system (dual boot: Windows 2000 or Windows 98 OSR 2), 800MHz AMD processor, 256MB PC133 RAM, and Creative Labs 3D Blaster GeForce 3 Titanium 200 64MB 3D accelerator video card.

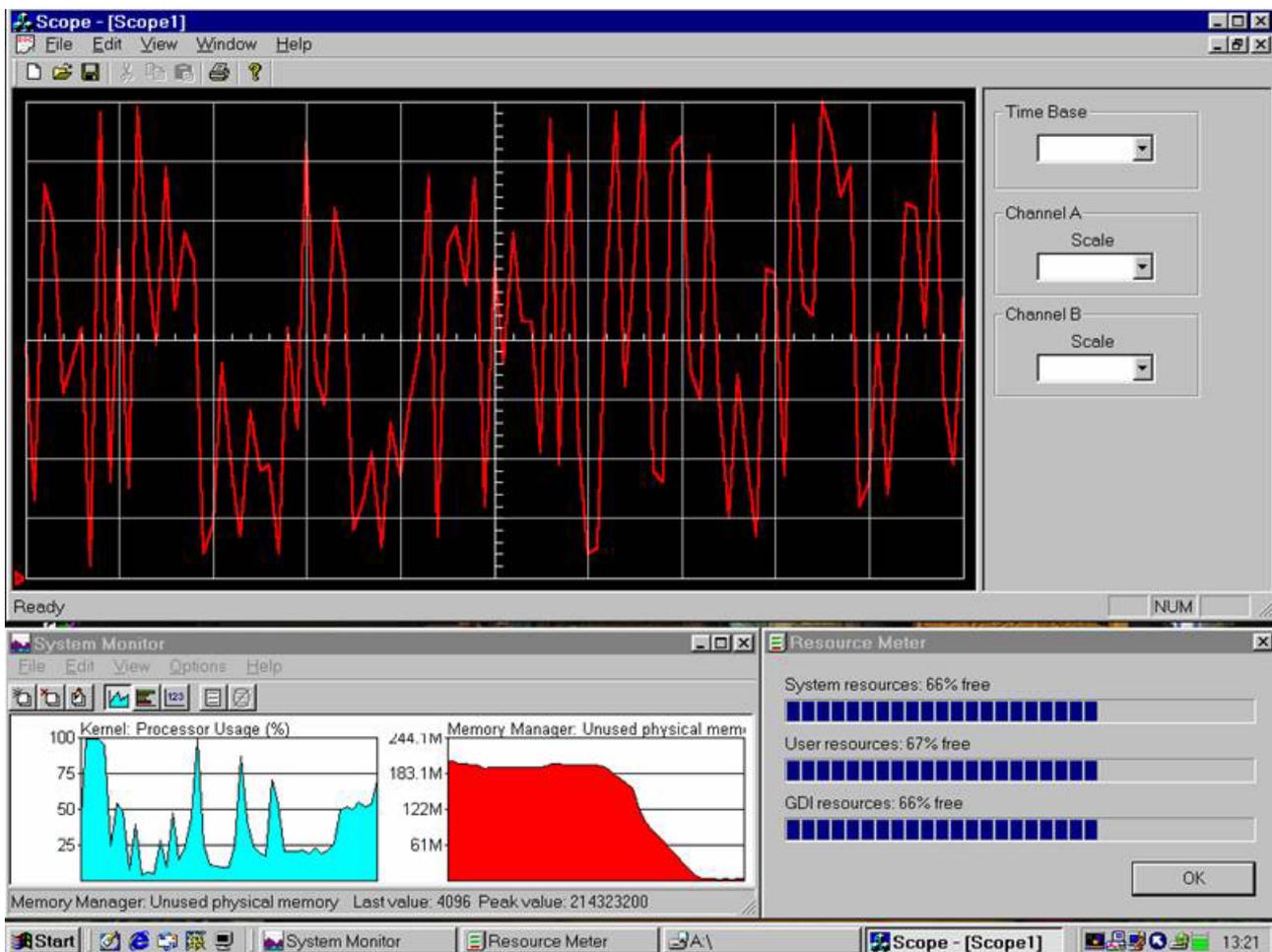


Figure 7.12a: Screen dump of 'System Monitor', 'Resource Meter' and 'Scope' V1.008a running in Windows 98

Figure 7.12a clearly shows that there is a problem; CPU load is high, systems resources are dropping fast and free physical memory has been completely used up. Note that this screen dump was taken after 10 seconds, the system resources continue dropping; after a minute the system resources were less than 10% and the system was completely unusable.

The reason why the screen dump (figure 7.12a) was taken after 10 seconds was because the system resources got so low that it was impossible to capture the screen. Screen dump (figure 7.12b) was taken after 40 seconds, notice that it was impossible to get a colour image, after a minute it was completely impossible to capture the screen.

Clearly there must be a big memory leak problem.

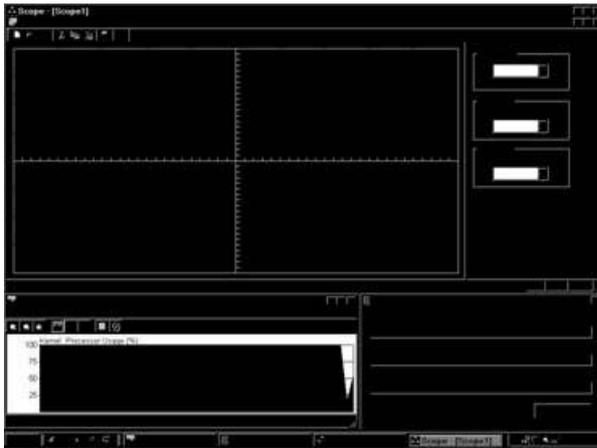


Figure 7.12b: Screen dump of Windows 98 running 'Scope' V1.008a taken after 40 seconds

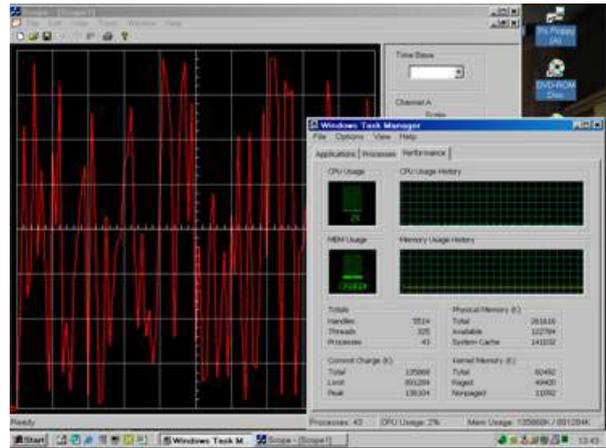


Figure 7.12c: Screen dump of Windows 2000 running 'Scope' V1.008a

Figure 7.12c clearly shows that Windows 2000 does not have any problems running scope V1.008a, the CPU load is less and 2% and the memory usage is stable. At first it was not understood why it was working in Windows 2000 and not Windows 98.

On the 24th December 2001 the problem was solved, there was indeed a memory leak. Every 100ms the screen is refreshed, e.g. a new bitmap is created and the waveform is drawn to the bitmap, this bitmap is then drawn to the screen, but the existing screen bitmap was being lost in memory. Therefore 10-times a second, a bitmap the size of the graphical display was being lost in memory, eating-up memory resources.

The solution was to setup a pointer to this old bitmap, and delete the object.

```
void CScopeView::OnDrawCH1 ()
{
    ...
    CBitmap *m_bitmapOldCH1 = NULL;
    ...
    if (m_dcCH1.GetSafeHdc() == NULL) // if we don't have one yet, set up a memory dc for ch1
    {
        m_dcCH1.CreateCompatibleDC(&dc);
        m_bitmapCH1.CreateDiscardableBitmap(&dc, m_nClientWidth, m_nClientHeight);
        m_bitmapOldCH1 = m_dcCH1.SelectObject(&m_bitmapCH1);
    }
    else
    {
        m_bitmapCH1.CreateDiscardableBitmap(&dc, m_nClientWidth, m_nClientHeight);
        m_bitmapOldCH1 = m_dcCH1.SelectObject(&m_bitmapCH1);
    }

    m_bitmapOldCH1->DeleteObject(); // Make Sure there is not a memory leak
    ...
}
```

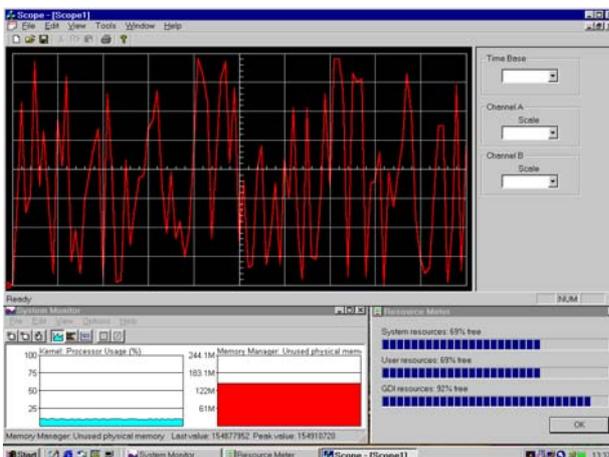


Figure 7.12d: Screen dump of Windows 98 running 'Scope' V1.010a

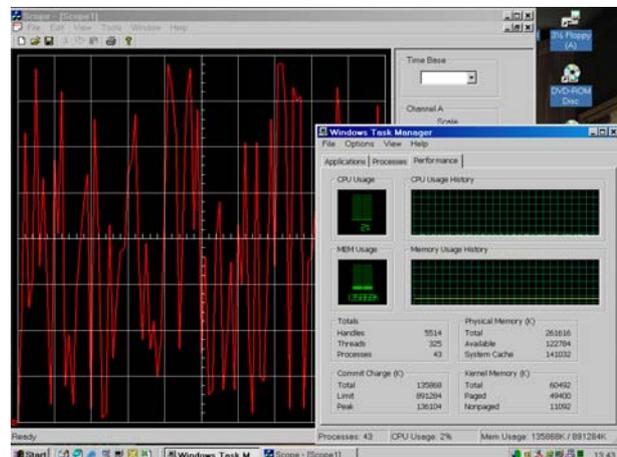


Figure 7.12e: Screen dump of Windows 2000 running 'Scope' V1.010a

Screen dump (figure 7.12d) clearly shows that this solved the problem, CPU load is now 12%, and system resources (including physical memory) are stable. Figure 7.12e clearly shows that the modifications did not affect the operation of the 'scope' program in Windows 2000, CPU load is less than 2% and memory usage is stable.

The reason why 'Scope' version 1.008a worked in Windows 2000 is because it is a far superior operating system that can deal with memory leaks, while Windows 98 cannot deal with memory leaks (one of the reasons why it's so unstable). This experiment clearly demonstrates how much better Windows 2000 is than Windows 98, also notice that the CPU load is much lower in Windows 2000 (<2%) when compared with Windows 98 ($\approx 12\%$).

7.13. Windows XP Compatibility Problem (scope v1.017a, 05/02/2002)

On the 3rd of February 2002 it was discovered that there is a problem when running the scope program in Windows XP, CPU load was high, which increased when the scope display was enlarged. At first it was not understood why this could be happening as the program had no problems running in Windows 98 and Windows 2000. Note Windows XP running on laptop (AMD 1GHz, 8MB 3D video card, 256MB of PC133 RAM) and Windows 2000/98 dual boot running on desktop (AMD 800MHz, 64MB 3D video card, 256MB of PC133 RAM).

The colours settings in the display control panel (Win XP) was changed from 24-bit to 16-bit, this reduced CPU load. But in Windows 2000 the colours are set to 32-bit and the CPU load is less than 2%, and in Windows 98 the colours are also set to 32-bit and the CPU load is less than 10%. Clearly Windows XP is trying to display the scope graphics at full colour specified in the display control panel.

The program creates a new bitmap (in XP same number of colours as the system) every time the screen is refreshed and deletes the old bitmap to make sure there is not a memory leak. Clearly this is bad programming practice as it is not necessary to create a new bitmap every time, but only when the screen is resized. The code will be modified in the near future so that this is the case, this will reduce CPU load but the problem remains when the window is being resized. It makes sense to try and also modify the code so that only a 256 colour bitmap is created, and not one with same number of colours as the system, in reality the scope display will use less than 16 colours (no need for 16-bit/24-bit/32-bit colour).

The big question is why this problem is not present in Windows 2000, there are two possible reasons: -

1. Windows 2000 may limit the number of colours that can be used by the program, e.g. the scope display is not 32-bit colour as used by the system.
2. The high end Creative Labs 3D Blaster GeForce 3 Titanium 200 64MB (400MHz DDR RAM) 3D accelerator video card in the desktop (running dual boot Windows 98/Windows 2000) is able to do some of the work. Although the scope program is not using DirectX or OpenGL hence this is unlikely.



Figure 7.13a. Screen dump of Compatibility option in Win XP

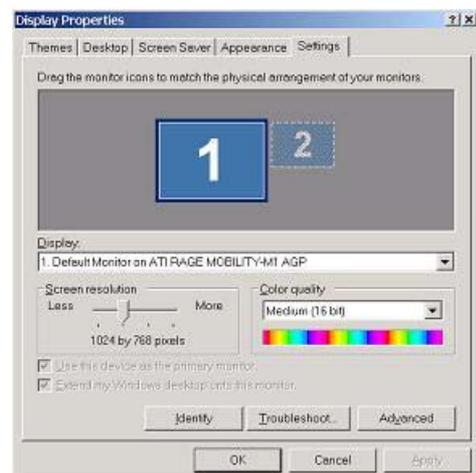


Figure 7.13b. Screen dump of display properties in Win XP

Figure 7.13a shows the compatibility options available for the scope program running in Windows XP; notice the “Run in 256 colours” option. If this option is clicked the program is limited to 256 colours (this works). Figure 7.13b shows the display properties in Window XP, note when running in 16-bit colour mode the scope program runs smoothly.

7.14. Additional Video Card Features That May Explain Low CPU Usage

The scope program when running on the desktop system running Windows 2000 (800MHz AMD processor, 256MB PC133 RAM, and Creative Labs 3D Blaster GeForce 3 Titanium 200 64MB 3D accelerator video card) uses less than 2% of the CPU load. But when tested on a UJ computer (lab: 6C49, computer: Intel P4 1.5GHz, 256MB RAM, Windows 2000), it was shocking to note that the scope program was using 10% of the CPU, that's 5-times more than that of the 800Mhz desktop system even though its clock speed is almost half the speed. OK AMD processors are faster than Intel's (e.g. 800MHz AMD is equal to 1GHz Intel P4), but not by this amount so something else must be making up the difference.

Clearly the video card must be accelerating windows graphics as well as direct X, and open GL and may be the reason for the low CPU usage on the desktop, when compared to the laptop and university machines. It was noticed that Geforce 3 video card drivers have added new features to windows that makes it probable that the video card is hardware accelerating standard windows operations.

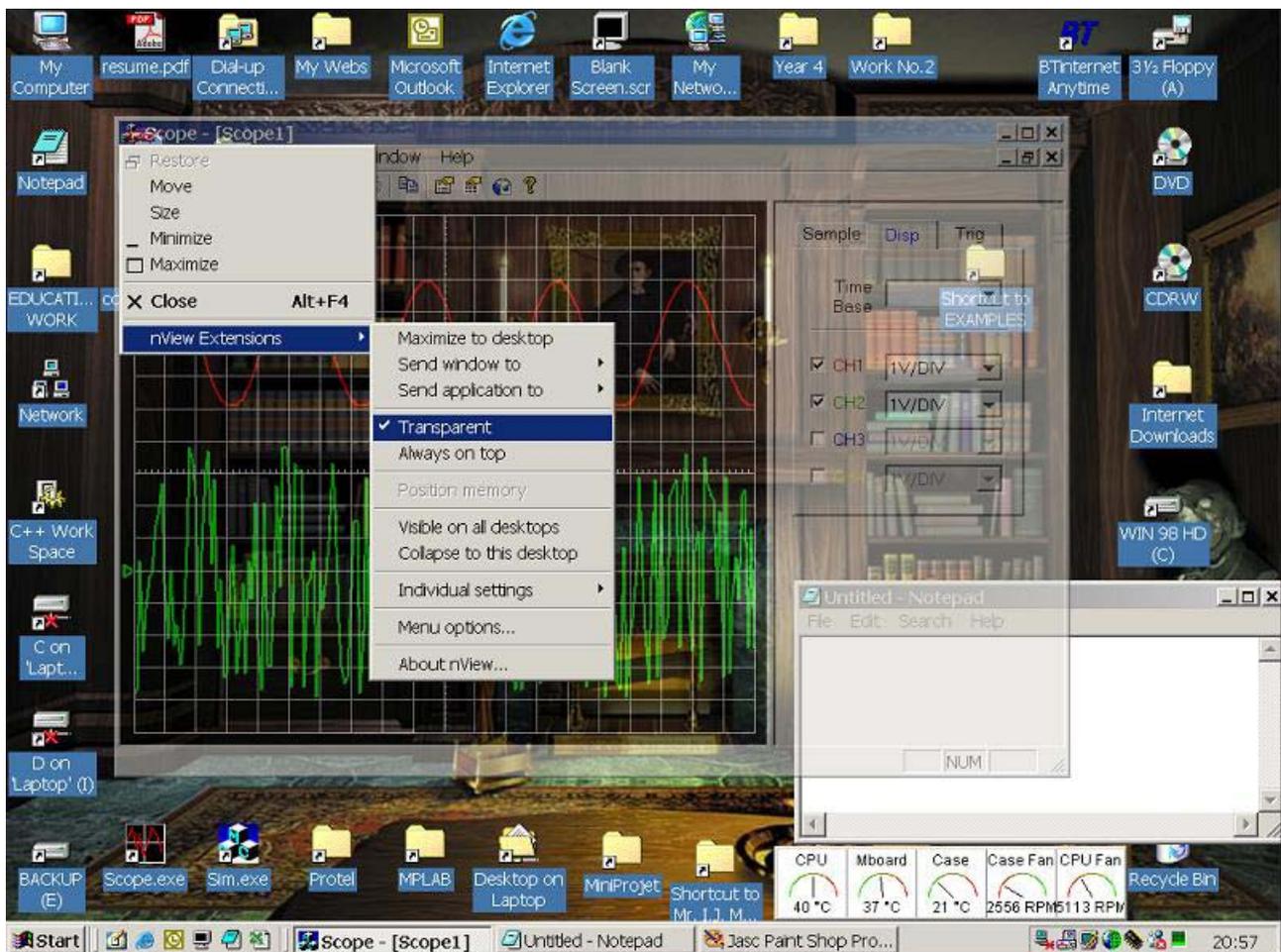


Figure 7.14a. Demonstrating the transparent window feature, added to the windows system menu by the video card drivers.

Figure 7.14a demonstrates the transparent window feature that was added to the windows system menu by video card drivers. Clearly this feature requires hardware acceleration; hence it is plausible that all windows based drawing is being acceleration by hardware.

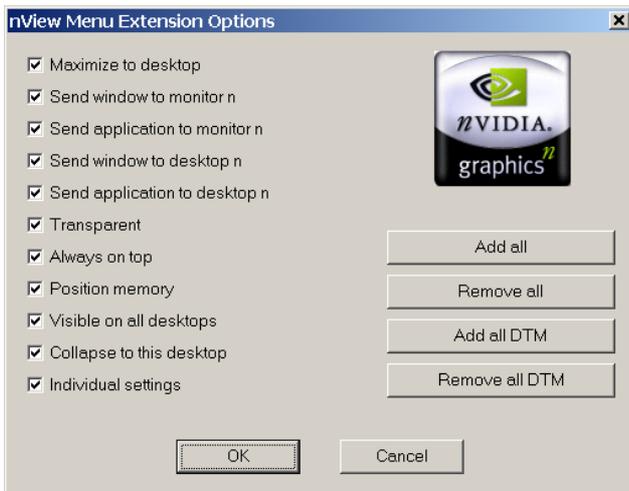


Figure 7.14b. nView Menu Extension Options

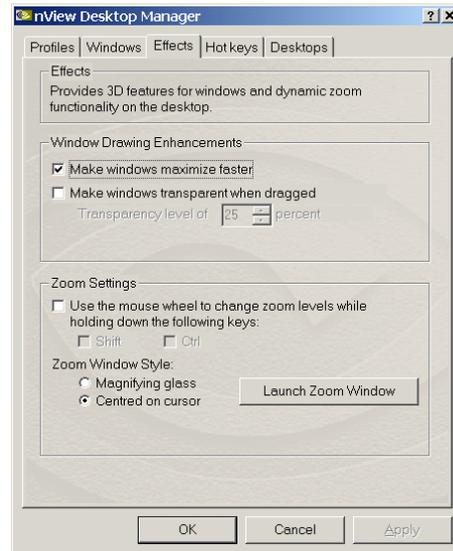


Figure 7.14c. nView Desktop Manager

Clearly the scope program is far too processor hungry and future modification is required to make it more efficient. Allow if 10% of the processor is used for a 1.5GHz machine, this means the program should run on a 150 MHz machine (perhaps poorly). By reducing the refresh rate of the scope display (or using a high-end video card) the scope program could be ran on slower machines, it has been tested on a p90 and runs smoothly with a refresh rate of 200 milliseconds, but the CPU load is at 100% and scope controls are sluggish to react to user input.

8.0. THE SIMULATOR PROGRAM

This program is a windows dialog based program written in Microsoft Visual C++ 6.0, Figure 8.0a shows a screen dump of the main dialog. The program is easy to use; the user selects the type of waveform, amplitude and frequency for each channel. The PC based scope program will automatically enable / disable channels using the 'control' protocol (user can override using the tick boxes beside channel number).

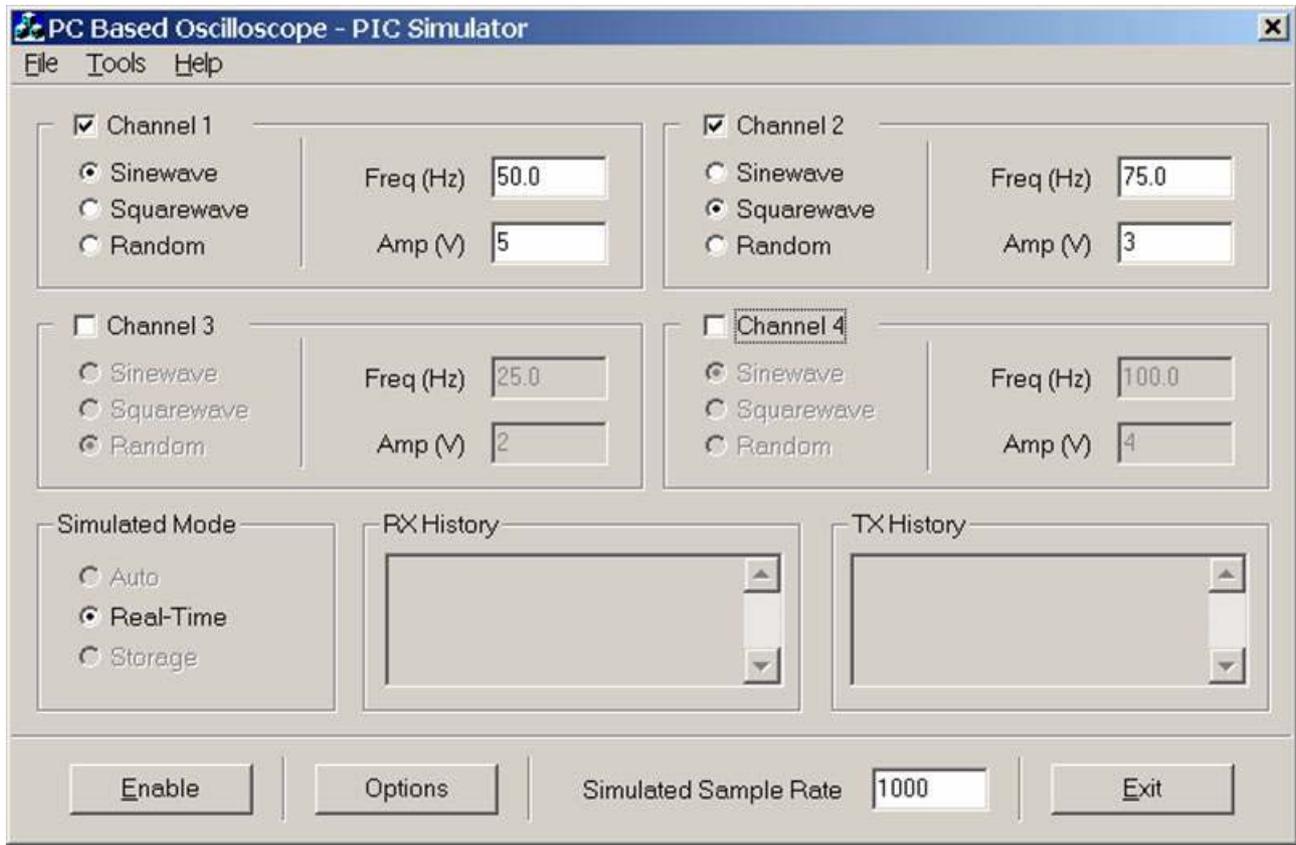


Figure 8.0a. Screen dump of sim.exe V1.007a - 29/03/2002

If a channel is disabled all of its controls are greyed (i.e. ch3 and ch4 are disabled in figure 8.0a), notice that the user has the option to specify real-time, storage or auto (controlled by PC). At present 'auto' and 'storage' mode have not been implemented. There are two large edit boxes that were designed to be used to display RX and TX history in a low level format, for debugging purposes, this feature was not implemented because there were no communication problems during development.

The program was used to test the communication protocols (real-time, storage, control) and graphical scope display (including triggering methods) at different baud rates. Note accurate timing is not simulated as this is difficult to achieve, hence the best way of testing the timing is on an actual PIC (hence the frequency settings for each channel are to be used as a guide and are not accurate).

Note the amplitude value of each channel is between 0 to 5V assuming this range is selected on the scope program (e.g. input range). It was foreseen that the scope program would control the input range, but this has not been implemented (shortage of time, future development), hence for the amplitude values on the simulator program to be accurate the correct range must be selected. This program can still be used to test other voltage input ranges, for example the simulator program 5V represents an ADC reading of 1024, and 0V represents an ADC reading of 0, hence if -10 to 10 V range on the scope program is selected 5V will represent 10V, 2.5V will represent 0V and 0V will represent -10V.

Simulation of all four channels is possible, after the user has setup each channel, communication with the scope program is enabled by clicking the [Enable] button. The user can even specify a simulated sample rate, basically specifying how many real-time frames are transmitted per second.

Figure 8.0b shows a screen dump of the 'Options' dialog box, which is used to configure the RS232 serial port in a user-friendly manner.

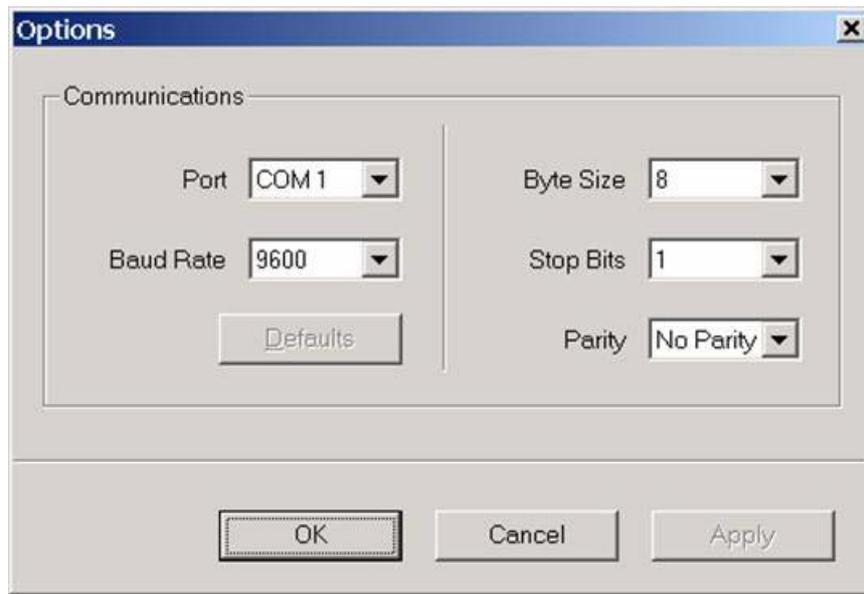


Figure 8.0b. Screen dump of options dialog in sim.exe V1.007a – 29/03/2002

Figure 8.0c shows a screen dump of the 'About' dialog box, which displays information about the program and the author, including some useful hyperlinks.

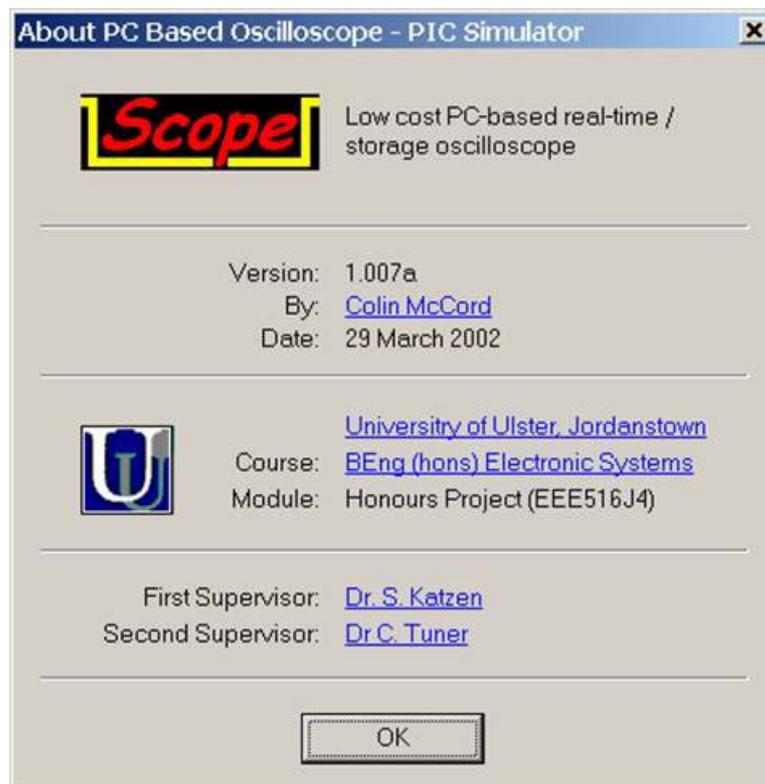


Figure 8.0c. Screen dump of about dialog in sim.exe V1.007a – 29/03/2002

8.1. How Channel Simulation Is Achieved?

A real-time software interrupt is used to call the communication protocol function 'ComProtocolTop()'. When an interrupt occurs the program first checks that communications is enabled, if its enabled function

'ComProtocolTop()' is called, then it sets the timer (delay until next interrupt) based on the specified 'simulated sample rate', so that the specified number of samples per second is actually transmitted per second.

This function is called on software interrupt: -

```
void CSimDlg::OnTimer(UINT nIDEvent)
{
    double timer,freq;

    if(nIDEvent == ID_TIMER_COMS)
    {
        KillTimer(ID_TIMER_COMS);
        if (m_Enable == true) ComProtocolTop();

        /** Set the timer based on the value in the sample rate edit box */
        freq = (double)GetDlgItemInt(IDC_SAMPLERATE);
        timer = (1.0/freq)*10000.0;
        SetTimer(ID_TIMER_COMS, (int)timer,NULL);
    }

    ...

    CDialog::OnTimer(nIDEvent);
}
```

This function is called by the OnTimer function: -

```
void CSimDlg::ComProtocolTop()
{
    int i;
    UpdateData(TRUE);

    /** RealTime **/
    if (IsDlgButtonChecked(IDC_REALTIME))
    {
        for (i = 1; i<10;i++)
        {
            if (m_EnableCH1 == TRUE)    SimCH1();
            if (m_EnableCH2 == TRUE)    SimCH2();
            if (m_EnableCH3 == TRUE)    SimCH3();
            if (m_EnableCH4 == TRUE)    SimCH4();
        }
    }
}
```

This function simulates CH1 (all four channels are done the same way): -

```
void CSimDlg::SimCH1()
{
    /** 15/03/2002, by CKM Version 1.6 */

    unsigned short temp;
    unsigned char ch1_b1, ch1_b2;

    double value;
    double freq = 50.0;
    double amp;

    amp = (double)GetDlgItemInt(IDC_AMP_CH1,NULL,TRUE);
    freq = (double)GetDlgItemInt(IDC_FREQ_CH1) * 200;

    if(m_nCH1WaveType == SINE)
    {
        value = amp * sin((2*3.14159 *freq *t1));
        temp = (unsigned short)(512.0+(102.4 * value));
        t1 = t1 + 1;
        if (t1 > freq) t1 = 1.0;
    }

    if(m_nCH1WaveType == SQUARE)
    {
        value = 10 * sin((2*3.14159 *freq *t1));
        if (value >0) value = amp;
    }
}
```

```
        else value = amp * -1.0;
        temp = (unsigned short)(512.0+(102.39 * value));
        t1 = t1 + 1;
        if (t1 > freq) t1 = 1.0;
    }

    if(m_nCH1WaveType == RANDOM)
    {
        temp = rand()%1024;
    }

    ch1_b1 = temp >> 5;
    ch1_b1 = ch1_b1 & 0x1F; // & 00011111

    ch1_b2 = (unsigned char)temp;
    ch1_b2 = ch1_b2 | 0x80; // | 1000 0000

    trans (ch1_b1);
    trans (ch1_b2);
}
```

9.0. TEST PROGRAM

Test.exe is a MS DOS based application designed to test the digital hardware (using RS232), it has many features and has a user-friendly interface, allow some of the test procedures are low-level designed for use by an engineer and not the average person. Full source code is found in appendix 2.

Note allow this is a DOS program it will run under Windows, even though Windows NT/2000/XP do not allow direct access to the hardware (e.g. RS232), it will simulate the operation of the RS232 UART allowing DOS based programs to access the serial port as long as another program is not already using the required port.

9.1. User Manual

Main menu, user has 9 options (just hit the key corresponding to the required function): -

```
PC-Based Oscilloscope Test Program
-----
RX Terminal          -- hit 1
TX Terminal          -- hit 2
Manual RTU frame with CRC -- hit 3
Manual RTU frame no CRC -- hit 4
Simulated Random Noise -- hit 5
Loop-back test      -- hit 6
Change Baud Rate    -- hit 7
About               -- hit 8
Quit                -- hit Q
```

[1] RX Terminal

```
RX Terminal
-----
Waiting for message (press a key to stop)
```

Simple RX Terminal, basically every character that is received from COM 1, is printed on the screen using the ASCII format. Note some non-alphanumerical characters have special functions, e.g. Backspace, Bell, etc... If [anykey] is pressed the program returns to the main menu.

This function can be used to test the PICs RS232 (transmit mode) and all of the hardware involved, e.g. PIC UART, MAX232 and cable. It can also be used to help debug the PIC software, e.g. printf() (CCS code) can be used to display variable information in order to try and work out what is happening.

[2] TX Terminal

```
TX Terminal
-----
Type your message hit [ENTER] to transmit: -
```

Simple TX terminal, the user types in a message and then hits the [ENTER] key to transmit the message through RS232. Note after transmission the 'RX Terminal' function is automatically called, so that any reply can be received.

This function can be used to test the PICs RS232 (receive mode) and all of the hardware involved, e.g. PIC UART, MAX232 and cable. Since 'RX terminal' is called automatically after transmission, it is possible to fully test RS232 communications, e.g. setup the PIC so that it transmits every character it receives. If the received message is the same as the transmitted message clearly RS232 communications is working.

[3] Manual RTU frame with CRC

```

Manual RTU frame with added CRC
-----
Enter RTU_frame in byte's enter 'DF' when finished
RTU_frame[0] (e.g 01) = 01
RTU_frame[1] (e.g 01) = 02
RTU_frame[2] (e.g 01) = df
Calculated CRC (Hex):      81E1
Total Frame Length (Dec): 4

Sending RTU Frame (HEX): -
1 2 81 E1

Finished Sending RTU Frame

```

User manually enters a frame of data in hexadecimal, once 0xDF is entered the program assumes that the frame is complete and calculates the 16-bit CRC which is added to the end of the frame. The whole frame is then transmitted through RS232.

This function can be used to generate test 'storage mode' frames when testing the scope program and test 'control protocol' frames for manually controlling the PIC.

[4] Manual RTU frame no CRC

```

Manual RTU frame no CRC
-----
Enter RTU_frame in byte's enter 'DF' when finished
RTU_frame[0] (e.g 01) = 1
RTU_frame[1] (e.g 01) = 2
RTU_frame[2] (e.g 01) = df
Total Frame Length (Dec): 2

Sending RTU Frame (HEX): -
1 2

Finished Sending RTU Frame

```

User manually enters a frame of data in hexadecimal, once 0xDF is entered the program assumes that the frame is complete and transmits the frame. This function can be used to generate test 'real-time' frames.

[5] Simulated Random Noise

```

Simulated Random Noise
-----
No delay between bytes                -- Hit 1
Random delay (0 to 100ms) between bytes -- hit 2
User sets delay in milliseconds       -- hit 3
Cancel - Return to Main Menu          -- hit 4

```

This function simulates random noise; there are three formats to choose from. The first transmits random bytes through RS232 continuously with no delay between the bytes. The second has a random delay between bytes and the third has a user configurable delay between the bytes.

This function is used to test the reliability of both the scope.exe program and the PIC embedded software. For example simulated noise is sent to the scope.exe program for a couple of hours, during this time the program must not crash and when the random noise has been removed the program must continue to operate normally.

[6] Loop-Back Test

```

Test Started: 11:02:23
-----
Testing          .... 100% - FAIL
* Test 1 - FAIL
* Test 2 - FAIL
* Test 3 - FAIL
-----
Test Finished: 11:02:25

```

Three tests are carried out: -

- Test 1: Data 0 → 255 → 0 is transmitted with a 16-bit CRC and compared with the received block (514 bytes).
- Test 2: Random data from 0 to 255 is transmitted with a 16-bit CRC and compared with the received block (514 bytes).
- Test 3: Random data 0x00 or 0xFF is transmitted with a 16-bit CRC and compared with the received block (514 bytes).

This function can test the PC serial port by connecting pin2 and pin3 together (RX & TX), or PIC software can be written to fully test communications by transmitting every character it receives. If this test is successful, it is certain that communications between the PC and the PIC is extremely good as this test requires an extremely good clean connection (will fail if one character has been corrupted).

[7] Change Baud Rate

```

Baud Rate (Current = 009600 bps)
-----
115200 bps -- bit 1
 57600 bps -- hit 2
 38400 bps -- hit 3
 19200 bps -- hit 4
 14400 bps -- hit 5
  9600 bps -- hit 6
  4800 bps -- hit 7
  2400 bps -- hit 8
  1200 bps -- hit 9
No Change  -- hit 0

```

User can change the baud rate by hitting the appropriate key.

[8] About

```

-----
| Final Year Project EEE516J4          |
| University of Ulster                 |
| BEng (hons) Electronic Systems      |
|                                     |
| Date:           22/02/2002          |
| Revision:       1.05                 |
| By:             Colin K McCord      |
|                                     |
-----

```

Information about the program and the author.

9.2. How is RS232 Communications Achieved?

Setup UART constants and RS232 buffer: -

```
#define COM 0x3f8      /* COM1=0x3f8, COM2=0x2f8 */
#define RBR 0         /* Receive Buffer Register */
#define THR 0         /* Transmit Holding Reg. */
#define DLL 0         /* Divisor Latch LSB */
#define DLM 1         /* Divisor Latch MSB */
#define IER 1         /* Interrupt Enable Reg. */
#define IIR 2         /* Interrupt ID Register */
#define FCR 2
#define AFR 2
#define LCR 3         /* Line Control Register */
#define MCR 4         /* Modem Control Register */
#define LSR 5         /* Line Status Register */
#define MSR 6         /* Modem Status Register */
#define SCR 7         /* The Scratch Register */
#define I8259M 0x21
#define I8259 0x20    /* The address of the 8259 */
#define EOI 0x20     /* The end of int command */

#define IRQ 4
#define SIZE 512     /* Buffer size */

char buffer[SIZE];  /* Receiving buffer */
char *i_get, *i_put; /* Pointer to buffer */
unsigned int baud;  /* Baud Rate */
```

Initialise UART (set speed, set data format, set ISR address etc...): -

```
int line,mask;      /* Comm parameters */

void interrupt(*old_handler)();
old_handler=getvect(IRQ+8); /* Old ISR address */

i_put=i_get=buffer; /* Pointer initialisation */
memset(buffer,0,SIZE); /* Buffer initialisation */

baud=0x0c;          /* 0x01 = 115200 bps
                    0x02 = 57600 bps
                    0x03 = 38400 bps
                    0x06 = 19200 bps
                    0x0c = 9600 bps */

line=0x03;         /* 8 bits, non parity, 1 stop */

/* initialisation */
outportb(COM+IER,0);
inportb(COM+LSR);
inportb(COM+MSR);
inportb(COM+IIR);
inportb(COM+RBR);

/* set speed */
outportb(COM+LCR, inportb(COM+LCR) | 0x80);
outportb(COM+DLM,baud>>8);
outportb(COM+DLL,baud&0xff);
outportb(COM+LCR,inportb(COM+LCR) & 0x7f);

/* set data format */
outportb(COM+LCR,line);
outportb(COM+MCR,0x0b);
outportb(COM+IER,0x01);

/* set ISR address */
disable();
setvect (IRQ+8,com_handler);
enable();

mask=inportb(I8259M);
mask &=~(1<<IRQ);
outportb(I8259M,mask);
```

Interrupt routine for storing incoming characters on the buffer: -

```
void interrupt com_handler()
```

```

{
    /* -----
    | Function:      com_handler
    | Description:   Interrupt's normal program operation when
    |               incoming character's are detected and
    |               store's them on the buffer.
    | Input:        none
    | Output:       Incoming character's stored on buffer
    | Return:       none
    |----- */

    *i_put++=inportb(COM+RBR);
    if(i_put==&buffer[SIZE]) i_put=buffer;
    outportb(I8259,EOI);
}

```

Transmitting a single character: -

```

void trans(int c)
{
    /* -----
    | Function:      trans
    | Description:   Wait's until UART is ready and then
    |               transmits a single character.
    | Input:        Character 'c'
    | Output:       Transmit 'c' through COM port using RS232
    | Return:       none
    |----- */

    while(!(inportb(COM+LSR) & 0x20));
    outportb(COM+THR,c);
}

```

Receiving a single character: -

```

int readchr (char *c)
{
    /* -----
    | Function:      readchr
    | Description:   Check's if character has been received,
    |               store's received character into 'c'
    | Input:        Address of character 'c'
    | Output:       'c' now contains received character
    | Return:       0 when character is present else -1
    |----- */

    if (i_get==i_put) return(-1);
    *c=*i_get++;
    if(i_get==&buffer[SIZE]) i_get=buffer;
    return (0);
}

```

Changing the baud rate after Initialisation: -

```

void baud_rate(void)
{
    /* -----
    | Function:      baud_rate(void)
    | Description:   User selects, baud rate via keyboard
    | Input:        keyboard
    | Output:       baud rate changed
    | Return:       none
    |----- */

    int rate;      /* Baud Rate */

    /* 0x01 = 115200 bps
    0x02 = 57600 bps
    0x03 = 38400 bps
    0x06 = 19200 bps
    0x08 = 14400 bps
    0x0c = 9600 bps */
    clrscr();

    rate = (115200 / baud);
    if(baud != 1)
        printf(" Baud Rate (Current = %.6u bps)",rate);
}

```

```

else
    printf("  Baud Rate (Current = 115200 bps)");
printf("\n-----");
printf("\n    115200 bps -- bit 1");
printf("\n    57600 bps -- hit 2");
printf("\n    38400 bps -- hit 3");
printf("\n    19200 bps -- hit 4");
printf("\n    14400 bps -- hit 5");
printf("\n     9600 bps -- hit 6");
printf("\n     4800 bps -- hit 7");
printf("\n     2400 bps -- hit 8");
printf("\n     1200 bps -- hit 9");
printf("\n    No Change  -- hit 0");

switch(getch())
{
    case '1':
        baud = 0x01;
        break;
    case '2':
        baud = 0x02;
        break;
    case '3':
        baud = 0x03;
        break;
    case '4':
        baud = 0x06;
        break;
    case '5':
        baud = 0x08;
        break;
    case '6':
        baud = 0x0c;
        break;
    case '7':
        baud = 0x18;
        break;
    case '8':
        baud = 0x30;
        break;
    case '9':
        baud = 0x60;
        break;
    default:
        clrscr();
        return;
}

/* set speed */
outportb(COM+LCR, inportb(COM+LCR) | 0x80);
outportb(COM+DLM, baud>>8);
outportb(COM+DLL, baud&0xff);
outportb(COM+LCR, inportb(COM+LCR) & 0x7f);

rate = 115200 / baud;
if(baud != 1) printf("\n\nBaud Rate Changed to %u bps",rate);
else         printf("\n\nBaud Rate Changed to 115200 bps",rate);
clrscr();
}

```

The source code is reasonable readable (lots of comments and partitioned into many functions), hence if the reader of this report wishes to find out how any other aspect of the program works see appendix #### for the complete source code.

10.0. PIC SOFTWARE DEVELOPMENT

The philosophy used during the development of the PIC code was to keep it simple, straightforward, comprehensible, and to a minimum. There are many small programs designed for testing the hardware and ideas, each program is labelled mark 1, 2, 3, etc... The end result is that the PIC code is gradually built up step-by-step, instead of writing the entire program at once. This ensures that operational results are obtained, as testing producers are carried out at each stage, while if the program was written all at once, there is little chance it will work and could prove difficult to debug.

The high-level programming language C was chosen and not the low-level assembly code normally associated with PIC programming. There are many advantages for using C including: ease of programming, ease of modification, reusability of code, use of standard functions (e.g. printf, getch, putc, etc...), etc... But there is one drawback, C code is much less efficient, for example typically code produced by the C compiler (CCS) is at least twice as large as that programmed in assembly. Since the PIC16F877 has a large program store (8k) and runs at 20MHz this drawback is not a problem.

10.1. Mark 1.c (Test RS232 Communications)

This program is extremely simple; basically it initialises RS232 communications and transmits "Testing..." once a second (see figure 10.1a). This program was used to test RS232 communications (transmit mode), including MAX232, PIC UART and cable. The PIC is connected to a PC which is running the DOS-based test program in RX terminal mode to receive the incoming characters.

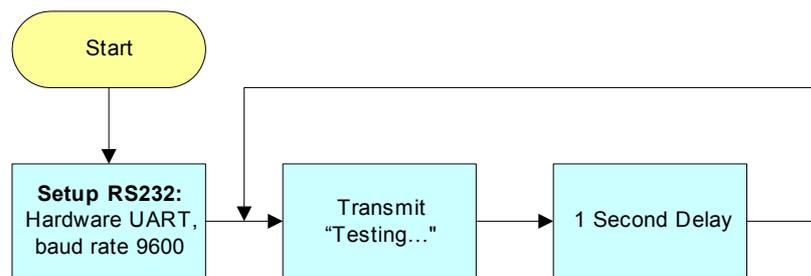


Figure 10.1a. Mark 1 flowchart

Mark 1.c source code: -

```

/* -----
| FILE      : mark 1.c
| PROJECT   : Low Cost PC Based Oscilloscope
| DESC      : Test RS232
| -----
| DATE      : 19/02/2002
| BY        : Colin K McCord
| VERSION   : 1.0
| -----
*/

#include <16F877.h>
#define PIC16F877 *:=16 ADC=10

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay(clock = 4000000) // 4MHz clock, change this value if using different clock speed.
#use rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

main()
{
    while(TRUE)
    {
        /* Note printf could be used but this function is wasteful and will not be used */
        putc('T'); // Transmit T
        putc('e'); // Transmit e
        putc('s'); // Transmit s
        putc('t'); // Transmit t
        putc('i'); // Transmit i
        putc('n'); // Transmit n
        putc('g'); // Transmit g
    }
}

```

```

    putc('.');    // Transmit .
    putc('.');    // Transmit .
    putc('.');    // Transmit .

    delay_ms(1000);    // Preset delay, repeat every second
}

```

10.2. Mark 2.c (Fully Test RS232 Communications)

This program is extremely simple; basically it initialises RS232 communications and waits for an incoming character, once a character is received the character is transmitted, this process repeats forever (see figure 10.2a). This program is used to fully test RS232 communications, using the test program in TX terminal mode to transmit a message and then in RX terminal mode to receive a message. A more comprehensive test can be carried out using the test program in the loop-back test mode; if this test passes it is certain that RS232 communications are optimal.

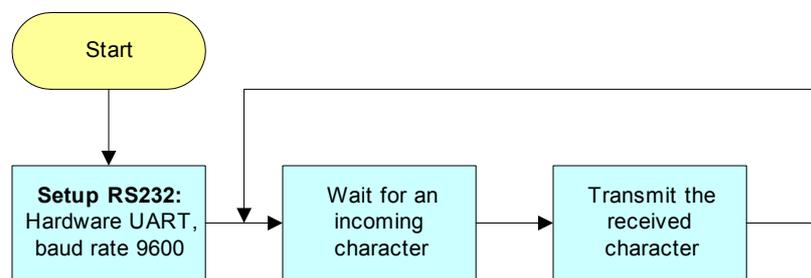


Figure 10.2a. Mark 2 flowchart

Mark 2.c source code: -

```

/* -----
| FILE      : mark_2.c
| PROJECT   : Low Cost PC Based Oscilloscope
| DESC      : Test Serial Communications, waits for an
|             incoming char and transmits the char.
| =====
| DATE      : 19/02/2002
| BY        : Colin K McCord
| VERSION   : 1.0
| ----- */

#include <16F877.h>
#define PIC16F877 *:=16 ADC=10

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay(clock = 4000000)
#use rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

main()
{
    /* Note kbhit() can be used to check for incoming char before calling getc() */
    while(TRUE)
    {
        putc(getc()); // Wait for incoming char and transmit...
    }
}

```

10.3. Mark 3.c (Real-Time Mode: 1 Channel, Fixed Sampling Delay)

This program reads ADC channel AN0 and transmits the reading through RS232 using the real-time frame structure with a fixed sampling delay of 10mS (that's a sample rate of 100Hz) before repeating (see figure 10.3a). This program is extremely useful, as it tests the ADC, the scope program, and the real-time frame structure. The sample rate is fixed; allow the sample delay can be manually modified (e.g. 1ms = 1000Hz) and the program recompiled and load into the PIC, hence testing of different sample rates is possible.

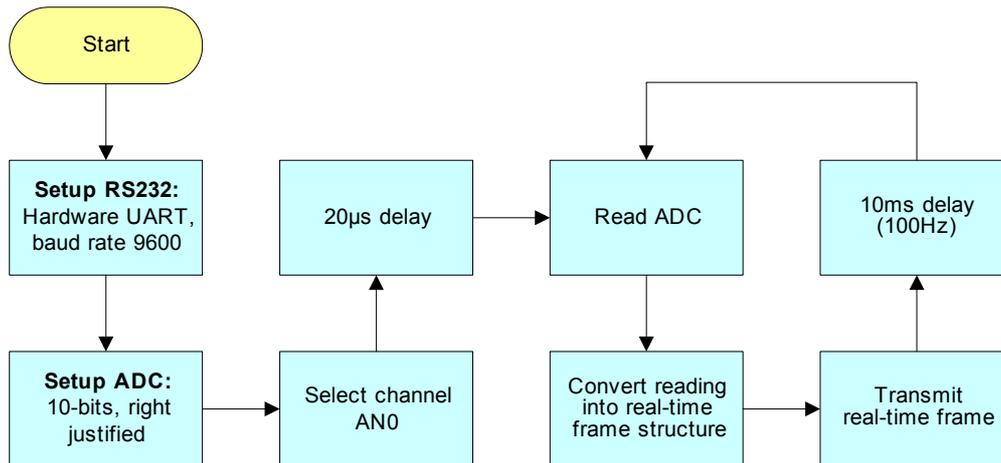


Figure 10.3a. Mark 3 flowchart

Mark 3.c source code: -

```

/* -----
| FILE      : mark_3.c
| PROJECT   : Low Cost PC Based Oscilloscope
| DESC     : Read CH1 ADC, transmit result through
|           : RS232 using the real-time frame format.
| -----
| DATE      : 19/02/2002
| BY       : Colin K McCord
| VERSION  : 1.1
| ----- */

#include <16F877.h>
#define PIC16F877 *:=16 ADC=10

// use #device adc = 10 to implement a 10-bit conversion,
// otherwise the default is 8-bits.

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay(clock = 4000000)
#use rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

main()
{
    // NOTE: by default in CCS all var's are unsigned
    long int adcValue;           // 16-bit storage for ADC reading
    char adcHI,adcLO;           // 8-bit storage for real-time frames.

    setup_adc_ports(A_ANALOG);   // RA0 - RA4 Analogue, RE0 - RE2 digital
    setup_adc(ADC_CLOCK_INTERNAL); // Use internal ADC clock.
    set_adc_channel(0);

    delay_us(20); // Delay for sampling cap to charge

    while(TRUE)
    {
        adcValue = read_adc(); // Get ADC reading

        /* Convert 16-bit adcValue to Real-time frame structure, CH1 */
        adcHI = (char)((adcValue >> 5) & 0x1f); // 0|0|0|d9|d8|d7|d6|d5
        adcLO = (char)((adcValue & 0x1f) | 0x80); // 1|0|0|d4|d3|d2|d1|d0

        putc(adcHI); // Transmit Byte 1 (d9...d5)
        putc(adcLO); // Transmit Byte 2 (d4...d0)

        delay_ms(10); // Preset delay, repeat every 10ms
    }
}

```

10.4. Mark 4.c (Test RS232, Baud-Rate Set by Dip-Switches)

Same as mark 1.c with the addition of function SetBaudRate(), this function sets the UART baud-rate based on the positions of the DIP switches that are connected to port E. This program is used to test the reading of the DIP switches and RS232 communications at different baud rates. Note that the PIC must be running at 20MHz because the percentage error for 115Kbps is too large at slower clock speeds.

Mark 4.c source code: -

```

/* -----
| FILE      : mark_4.c
| PROJECT   : Low Cost PC Based Oscilloscope
| DESC      : Test RS232, baud rate set by dip switches.
| =====
| DATE      : 01/03/2002
| BY        : Colin K McCord
| VERSION   : 1.0
| ----- */

#include <16F877.h>
#device PIC16F877 *=16 ADC=10

#fuses HS,NOWDT,NOPROTECT,NOLVP

#byte  PORTE = 0x09 // PortE lives in File 9

/* Note 20MHz clock must be used for 115,000 bps, the % error is to large at slower speeds */
#use delay(clock = 20000000) // 20MHz clock, change this value if using different clock speed.
#use rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

#use fast_io(E) // Fast access to PortE (don't fiddle with TRISE)

/* Forward declaration of functions */
void SetBaudRate();

main()
{
    set_tris_e(0x17); // TRISE = 00010111; RE2,RE1 and RE0 TTL Inputs

    SetBaudRate();

    while(TRUE)
    {
        /* Note printf could be used but this function is wasteful and will not be used */
        putc('T'); // Transmit T
        putc('e'); // Transmit e
        putc('s'); // Transmit s
        putc('t'); // Transmit t
        putc('i'); // Transmit i
        putc('n'); // Transmit n
        putc('g'); // Transmit g
        putc('.'); // Transmit .
        putc('.'); // Transmit .
        putc('.'); // Transmit .

        delay_ms(1000); // Preset delay, repeat every second
    }
}

void SetBaudRate()
{
    switch(PORTE & 0x07) // Read dip switches and setup baud rate
    {
        case 0: set_uart_speed(4800); break;
        case 1: set_uart_speed(9600); break;
        case 2: set_uart_speed(14400); break;
        case 3: set_uart_speed(19200); break;
        case 4: set_uart_speed(32768); break;
        case 5: set_uart_speed(38400); break;
        case 6: set_uart_speed(57600); break;
        case 7: set_uart_speed(115200); break;
    }
}

```

10.5. Mark 5.c (Fully Test RS232 Communications, Adjustable Baud-Rate)

Same as mark2.c with the addition of function SetBaudRate(), this function sets the UART baud-rate based on the positions of the DIP switches that are connected to port E. This program is used to fully test RS232 communications at different baud rates.

Mark 5.c source code: -

```

/* -----
| FILE      : mark_5.c
| PROJECT   : Low Cost PC Based Oscilloscope
| DESC      : Test Serial Communications, waits for an
|             incoming char and transmits the char. Baud
|             rate set by dip switches.
| -----
| DATE      : 01/03/2002
| BY        : Colin K McCord
| VERSION   : 1.0
| ----- */

#include <16F877.h>
#define PIC16F877 *16 ADC=10

#fuses HS,NOWDT,NOPROTECT,NOLVP

#byte  PORTE = 0x09    // PortE lives in File 9

/* Note 20MHz clock must be used for 115,000 bps, the % error is to large at slower speeds */
#define delay(clock = 2000000) // 20MHz clock.
#define rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

/* Forward declaration of functions */
void SetBaudRate();

main()
{
    set_tris_e(0x17);    // TRISE = 00010111; RE2,RE1 and RE0 TTL Inputs
    SetBaudRate();

    /* Note kbhit() can be used to check for incoming char before calling getc() */
    while(TRUE)
    {
        putc(getc()); // Wait for incoming char and transmit....
    }
}

void SetBaudRate()
{
    switch(PORTE & 0x07) // Read dip switches and setup baud rate
    {
        case 0: set_uart_speed(4800); break;
        case 1: set_uart_speed(9600); break;
        case 2: set_uart_speed(14400); break;
        case 3: set_uart_speed(19200); break;
        case 4: set_uart_speed(32768); break;
        case 5: set_uart_speed(38400); break;
        case 6: set_uart_speed(57600); break;
        case 7: set_uart_speed(115200); break;
    }
}

```

10.6. Mark 6.c (Real-Time Mode: 1 Channel, Fixed Delay, Adjustable Baud-Rate)

Same as mark3.c with the addition of function SetBaudRate(), this function sets the UART baud-rate based on the positions of the DIP switches that are connected to port E. This program is used to test the real-time communications protocol and scope program at different baud rates.

Mark 6.c source code: -

```

/*-----
| FILE      : mark_6.c                               |
| PROJECT   : Low Cost PC Based Oscilloscope         |
| DESC     : Read CH1 ADC, transmit result through   |
|           : RS232 using the real-time frame format. |
|           : Baud rate set by dip-switches.         |
|-----
| DATE      : 01/03/2002                             |
| BY        : Colin K McCord                         |
| VERSION   : 1.2                                    |
|----- */

#include <16F877.h>
#define PIC16F877 *16 ADC=10

// use #device adc = 10 to implement a 10-bit conversion,
// otherwise the default is 8-bits.

#fuses HS,NOWDT,NOPROTECT,NOLVP

#byte  PORTE = 0x09 // PortE lives in File 9

/* Note 20MHz clock must be used for 115,000 bps, the % error is to large at slower speeds */
#use delay(clock = 20000000)
#use rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

#use fast_io(E) // Fast access to PortE (don't fiddle with TRISE)

/* Forward declaration of functions */
void SetBaudRate();

main()
{
    // NOTE: by default in CCS all var's are unsigned
    long int adcValue; // 16-bit storage for ADC reading
    char adcHI,adcLO; // 8-bit storage for real-time frames.
    set_tris_e(0x17); // TRISE = 00010111; RE2,RE1 and RE0 TTL Inputs
    SetBaudRate();

    setup_adc_ports(A_ANALOG); // RA0 - RA4 Analogue, RE0 - RE2 digital
    setup_adc(ADC_CLOCK_INTERNAL); // Use internal ADC clock.
    set_adc_channel(0);

    delay_us(20); // Delay for sampling cap to charge

    while(TRUE)
    {
        adcValue = read_adc(); // Get ADC reading

        /* Convert 16-bit adcValue to Real-time frame structure, CH1 */
        adcHI = (char)((adcValue >> 5) & 0x1f); // 0|0|0|d9|d8|d7|d6|d5
        adcLO = (char)((adcValue & 0x1f) | 0x80); // 1|0|0|d4|d3|d2|d1|d0

        putc(adcHI); // Transmit Byte 1 (d9...d5)
        putc(adcLO); // Transmit Byte 2 (d4...d0)

        delay_ms(1); // Preset delay, repeat every 1ms, that's 1000 Hz
                    // baud rate must be at least 20000bps, try 32768bps.
    }
}

void SetBaudRate()
{
    switch(PORTE & 0x07) // Read dip switches and setup baud rate
    {
        case 0: set_uart_speed(4800); break;
        case 1: set_uart_speed(9600); break;
        case 2: set_uart_speed(14400); break;
        case 3: set_uart_speed(19200); break;
        case 4: set_uart_speed(32768); break;
        case 5: set_uart_speed(38400); break;
        case 6: set_uart_speed(57600); break;
        case 7: set_uart_speed(115200); break;
    }
}

```

10.7. Mark 7.c (Test Timer Interrupts)

This program uses all of the PICs built-in timers, basically timers 0-2 are setup to cause an interrupt. An interrupt subroutine has been written for each timer; at preset intervals (0.5s, 0.25s, & 0.1s) a message is transmitted to the PC. The main program is stuck in a loop transmitting "main..." every second, Timer0_ISR transmits "Interrupt_T0..." every 0.5s, Timer1_ISR transmits "Interrupt_T1" every 0.25s and Timer2_ISR transmits "Interrupt_T2" 10-times a second. These messages are received using a PC running the DOS based test program in RX terminal mode, hence testing of the PIC timer interrupts including crude timing analysis is achieved, for example there should be two "Interrupt_T0..." messages between every "Main...". See figure 10.7a for a simplified flowchart of the program.

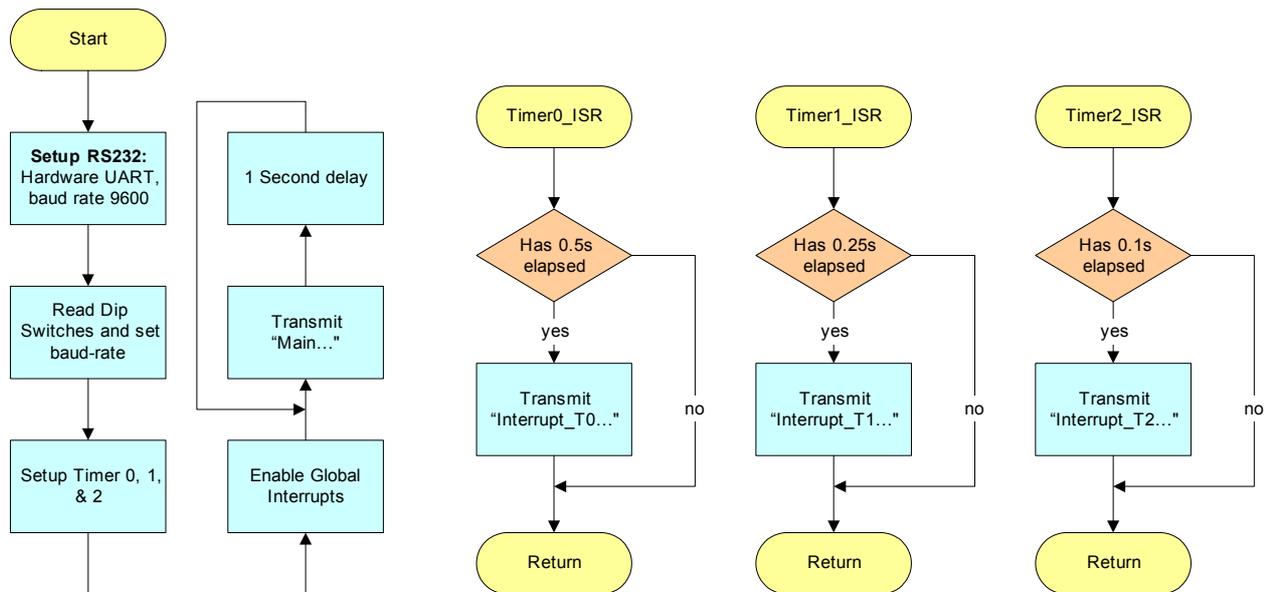


Figure 10.7a. Mark 7 flowchart

Mark 7.c source code: -

```

/*-----
| FILE      : mark_7.c
| PROJECT   : Low Cost PC Based Oscilloscope
| DESC      : Test timer interrupts
|-----
| DATE      : 05/03/2002
| BY        : Colin K McCord
| VERSION   : 1.1
|----- */

#include <16F877.h>
#define PIC16F877 *16 ADC=10

#fuses HS,NOVDT,NOPROTECT,NOLVP

#byte PORTE = 0x09 // PortE lives in File 9

/* Note 20MHz clock must be used for 115,000 bps, the % error is to large at slower speeds */
#use delay(clock = 2000000)
#use rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

#use fast_io(E) // Fast access to PortE (don't fiddle with TRISE)

#define T0_INTS_PER_SEC 76 // (20,000,000/(4*256*256))
#define T1_INTS_PER_SEC 76 // (20,000,000/(4*1*65536))

byte int_count0; // Number of T0 interrupts left before a 0.5s has elapsed.
byte int_count1; // Number of T1 interrupts left before a 0.25s has elapsed.
byte int_count2; // Number of T2 interrupts left before a 100 msec has elapsed.

#int_rtcc // RTCC (timer0) interrupt, called every time RTCC overflows (255->0)
Timer0_ISR()
{

```

```

        if(--int_count0==0)
        {
            printf("Interrupt_T0...");           // Every 0.5 seconds.
            int_count0 = T0_INTS_PER_SEC/2;
        }
    }

#INT_TIMER1           // timer1 interrupt, called every time timer1 overflows (65536->0)
Timer1_ISR()
{
    if(--int_count1==0)
    {
        printf("Interrupt_T1...");           // Every 0.25 seconds.
        int_count1 = T1_INTS_PER_SEC/4;
    }
}

#INT_TIMER2           // timer2 interrupt
Timer2_ISR()
{
    if(--int_count2==0)
    {
        printf("Interrupt_T2...");           // Every 0.1 seconds.
        int_count2 = 100;
    }
}

/* Forward declaration of functions */
void SetBaudRate();

main()
{
    set_tris_e(0x17);           // TRISE = 00010111; RE2,RE1 and RE0 TTL Inputs
    SetBaudRate();

    /** Setup timer0 (RTCC) **/
    int_count0 = T0_INTS_PER_SEC/2;           // 0.5 seconds
    set_rtcc(0);
    setup_counters(RTCC_INTERNAL,RTCC_DIV_256);
    enable_interrupts(RTCC_ZERO);

    /** Setup timer1 **/
    int_count1 = T1_INTS_PER_SEC/4;           //0.25 second
    setup_timer_1(T1_DIV_BY_1 | T1_INTERNAL);
    set_timer1(0);
    enable_interrupts(INT_TIMER1);

    /** Setup timer2 **/
    int_count2 = 100;           // 0.1 second
    setup_timer_2 (T2_DIV_BY_4,125,9);       // interrupt every 1ms
    set_timer2(0);
    enable_interrupts(INT_TIMER2);

    enable_interrupts(GLOBAL);

    while(TRUE)
    {
        printf("Main...");
        delay_ms(1000);           // 1 second delay
    }
}

void SetBaudRate()
{
    switch(PORTE & 0x07) // Read dip switches and setup baud rate
    {
        case 0: set_uart_speed(4800); break;
        case 1: set_uart_speed(9600); break;
        case 2: set_uart_speed(14400); break;
        case 3: set_uart_speed(19200); break;
        case 4: set_uart_speed(32768); break;
        case 5: set_uart_speed(38400); break;
        case 6: set_uart_speed(57600); break;
        case 7: set_uart_speed(115200); break;
    }
}

```

10.8. Mark 8.c (Real-Time Mode: Dual Channel, Fixed Delay, Adjustable Baud-Rate)

Same as mark6.c expect that instead of just sampling channel 1 the program chops between channel 1 and channel 2 (see figure 10.8a). This program is used to test the scope program in dual trace mode, triggering methods and stability of a trace when the other is being used as the trigger. Note the configuration of the ADC has been modified so that 32Tosc is used and not the internal RC oscillator. The reason for this change is that the PIC16F877 datasheet states "When the device frequencies are greater than 1MHz, the RC A/D conversion clock source is only recommended for sleep operation".

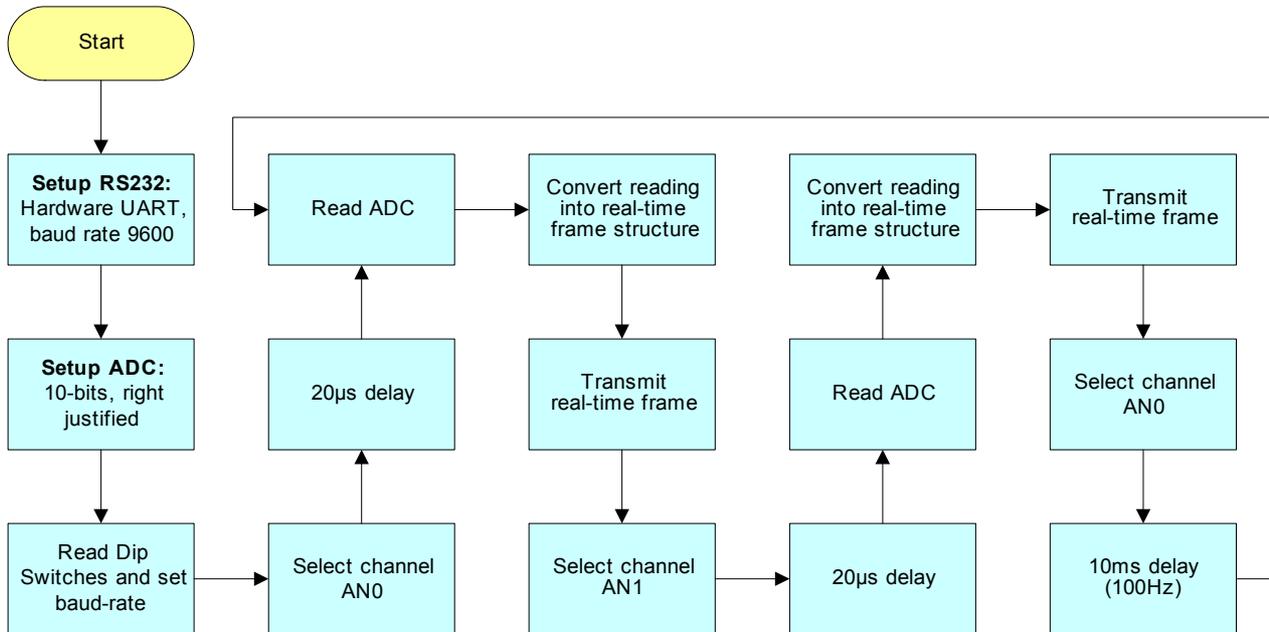


Figure 10.8a. Mark 8 flowchart

Mark 8.c source code: -

```

/* -----
| FILE      : mark_8.c
| PROJECT   : Low Cost PC Based Oscilloscope
| DESC      : Read CH1 & CH2, transmit result through
|            : RS232 using the real-time frame format.
|            : Baud rate set by dip-switches.
| -----
| DATE      : 05/03/2002
| BY        : Colin K McCord
| VERSION   : 1.4
| -----
*/

#include <16F877.h>
#define PIC16F877 *:=16 ADC=10

// use #device adc = 10 to implement a 10-bit conversion,
// otherwise the default is 8-bits.

#fuses HS,NOWDT,NOPROTECT,NOLVP

#byte  PORTE = 0x09 // PortE lives in File 9

/* Note 20MHz clock must be used for 115,000 bps, the % error is to large at slower speeds */
#use delay(clock = 20000000)
#use rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

#use fast_io(E) // Fast access to PortE (don't fiddle with TRISE)

/* Forward declaration of functions */
void SetBaudRate();

main()
{

```

```

// NOTE: by default in CCS all var's are unsigned
long int adcValue;           // 16-bit storage for ADC reading
char adcHI,adcLO;           // 8-bit storage for real-time frames.

set_tris_e(0x17);           // TRISE = 00010111; RE2,RE1 and RE0 TTL Inputs
SetBaudRate();

setup_adc_ports(A_ANALOG);   // RA0 - RA4 Analogue, RE0 - RE2 digital
setup_adc(ADC_CLOCK_DIV_32);
set_adc_channel(0);

delay_us(20); // Delay for sampling cap to charge

while(TRUE)
{
    adcValue = read_adc(); // Get ADC reading

    /* Convert 16-bit adcValue to Real-time frame structure, CH1 */
    adcHI = (char)((adcValue >> 5) & 0x1f); // 0|0|0|d9|d8|d7|d6|d5
    adcLO = (char)((adcValue & 0x1f) | 0x80); // 1|0|0|d4|d3|d2|d1|d0

    putc(adcHI); // Transmit Byte 1 (d9...d5)
    putc(adcLO); // Transmit Byte 2 (d4...d0)

    set_adc_channel(1);
    delay_us(20); // Delay for sampling cap to charge

    adcValue = read_adc(); // Get ADC reading

    /* Convert 16-bit adcValue to Real-time frame structure, CH2 */
    adcHI = (char)((adcValue >> 5) & 0x1f) | 0x20; // 0|0|1|d9|d8|d7|d6|d5
    adcLO = (char)((adcValue & 0x1f) | 0xA0); // 1|0|1|d4|d3|d2|d1|d0

    putc(adcHI); // Transmit Byte 1 (d9...d5)
    putc(adcLO); // Transmit Byte 2 (d4...d0)

    set_adc_channel(0);
    delay_ms(10); // Preset delay, repeat every 10ms, that's 100 Hz
}
}

void SetBaudRate()
{
    switch(PORTE & 0x07) // Read dip switches and setup baud rate
    {
        case 0: set_uart_speed(4800); break;
        case 1: set_uart_speed(9600); break;
        case 2: set_uart_speed(14400); break;
        case 3: set_uart_speed(19200); break;
        case 4: set_uart_speed(32768); break;
        case 5: set_uart_speed(38400); break;
        case 6: set_uart_speed(57600); break;
        case 7: set_uart_speed(115200); break;
    }
}

```

10.9. Mark 9.c (Real-Time Mode: Four Channels, Chop, Interrupt Time-Base)

This program uses timer 2 (PIC built-in timer) as the time-base, the timer causes an interrupt every 20 μ S. The variable 'TimeBaseMUX' is the time-base multiplier (e.g. 100 Hz = 10ms, hence TimeBaseMUX = 500). Note no sampling is done during the interrupt routine as all interrupts are disabled while an interrupt is being processed, hence the interrupt routine needs to take less than 20 μ S to execute or timing would be inaccurate (e.g. say the interrupt routine execution time was longer than 20 μ S, during this time timer 2 cannot cause any more interrupts, hence accurate timing is lost). The main program loops continuously checking the 'bSample' flag, if the flag is true the function Sample_RealTime() is called, else it continues to loop. The 'bSample' flag is set in the time-base interrupt routine, keeping the interrupt routine short.

Interrupt driven time-base has one big advantage over using preset delays: delay routines do not take into consideration processing delays (e.g. waiting for UART buffer to empty), while the PICs real-time timer will

continue to count no matter what the PIC is doing, and will cause an interrupt every 20µS (assuming interrupt routine is finished before it is time to called it again).

This program samples all four channels (chop mode). For example channel 1 is sampled, then channel 2, then channel 3 and then channel 4. See figure 10.9a for a simplified flowchart of the program.

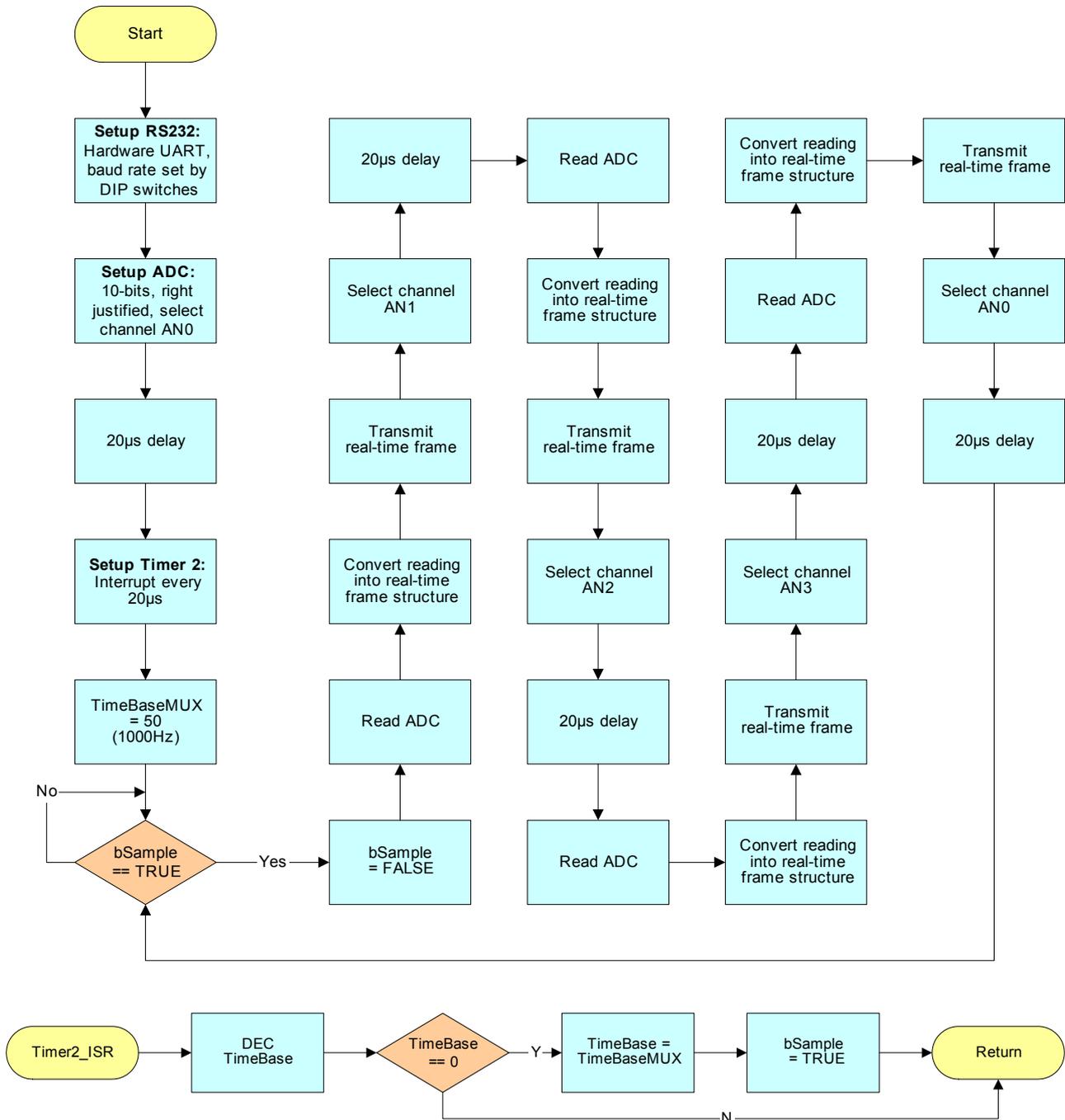


Figure 10.9a. Mark 9 flowchart

Mark 9.c source code: -

```

/*
| FILE      : mark_9.c
| PROJECT   : Low Cost PC Based Oscilloscope
| DESC      : real-time sampling using Interrupt
|           : timebase, time-base is fixed. All channels
|           : are sampled CH1 to CH4 chop mode.
|           : Baud rate set by dip-switches.
|=====
    
```

```

| DATE      : 20/03/2002          |
| BY        : Colin K McCord     |
| VERSION   : 1.5                |
----- */

#include <16F877.h>
#device PIC16F877 *=16 ADC=10

// use #device adc = 10 to implement a 10-bit conversion,
// otherwise the default is 8-bits.

#fuses HS,NOWDT,NOPROTECT,NOLVP,NOBROWNOUT

#byte PORTE = 0x09 // PortE lives in File 9

/* Note 20MHz clock must be used for 115,000 bps, the % error is to large at slower speeds */
#use delay(clock = 20000000)
#use rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

#use fast_io(E) // Fast access to PortE (don't fiddle with TRISE)

long int TimeBase;
long int TimeBaseMUX;
char bSample;

/* Forward declaration of functions */
void SetBaudRate();
void Sample_RealTime();

/** TimeBase Interrupt called every 20uS that's 50KHz MAX **/
#INT_TIMER2
Timer2_ISR()
{
    // Sample Frequency = TimeBaseMUX * 20uS, e.g. 100Hz (1mS) = 20uS * 500
    if (--TimeBase==0)
    {
        TimeBase = TimeBaseMUX;
        bSample = TRUE;
    }
}

main()
{
    bSample = FALSE;

    set_tris_e(0x17); // TRISE = 00010111; RE2,RE1 and RE0 TTL Inputs
    SetBaudRate();

    setup_adc_ports(A_ANALOG); // RA0 - RA4 Analog, RE0 - RE2 digital
    setup_adc(ADC_CLOCK_DIV_32);
    set_adc_channel(0);

    delay_us(20); // Delay for sampling cap to charge

    /** Setup Time-Base timer2 **/ // 1/(20,000,000/(4*1*100*1)) = 20uS
    setup_timer_2 (T2_DIV_BY_1,100,1); // interrupt every 20us, that's 50Khz max
    set_timer2(0);
    enable_interrupts(INT_TIMER2);

    TimeBaseMUX = 50; // 1000 Hz
    TimeBase = TimeBaseMUX; // e.g. 50000 / 100 for 100 Hz (thats 500 * 20us = 10mS)

    enable_interrupts(GLOBAL);

    while(TRUE)
    {
        if(bSample == TRUE)
        {
            bSample = FALSE;
            Sample_RealTime();
        }
    }
}

```

```

void Sample_RealTime()
{
    // 20/03/2002. verision 1.0. sample ch1 to ch4, Chop Mode

    // NOTE: by default in CCS all var's are unsigned
    long int adcValue;           // 16-bit storage for ADC reading
    char adcHI,adcLO;           // 8-bit storage for real-time frames.

    /*** Channel 1 ***/
    adcValue = read_adc(); // Get ADC reading

    /* Convert 16-bit adcValue to Real-time frame structure, CH1 */
    adcHI = (char)((adcValue >> 5) & 0x1f); // 0|0|0|d9|d8|d7|d6|d5
    adcLO = (char)((adcValue & 0x1f) | 0x80); // 1|0|0|d4|d3|d2|d1|d0

    putc(adcHI); // Transmit Byte 1 (d9...d5)
    putc(adcLO); // Transmit Byte 2 (d4...d0)

    /*** Channel 2 ***/
    set_adc_channel(1);
    delay_us(20); // Delay for sampling cap to charge

    adcValue = read_adc(); // Get ADC reading

    /* Convert 16-bit adcValue to Real-time frame structure, CH2 */
    adcHI = (char)(((adcValue >> 5) & 0x1f) | 0x20); // 0|0|1|d9|d8|d7|d6|d5
    adcLO = (char)((adcValue & 0x1f) | 0xA0); // 1|0|1|d4|d3|d2|d1|d0

    putc(adcHI); // Transmit Byte 1 (d9...d5)
    putc(adcLO); // Transmit Byte 2 (d4...d0)

    /*** Channel 3 ***/
    set_adc_channel(2);
    delay_us(20); // Delay for sampling cap to charge

    adcValue = read_adc(); // Get ADC reading

    /* Convert 16-bit adcValue to Real-time frame structure, CH3 */
    adcHI = (char)(((adcValue >> 5) & 0x1f) | 0x40); // 0|1|0|d9|d8|d7|d6|d5
    adcLO = (char)((adcValue & 0x1f) | 0xC0); // 1|1|0|d4|d3|d2|d1|d0

    putc(adcHI); // Transmit Byte 1 (d9...d5)
    putc(adcLO); // Transmit Byte 2 (d4...d0)

    /*** Channel 4 ***/
    set_adc_channel(3);
    delay_us(20); // Delay for sampling cap to charge

    adcValue = read_adc(); // Get ADC reading

    /* Convert 16-bit adcValue to Real-time frame structure, CH4 */
    adcHI = (char)(((adcValue >> 5) & 0x1f) | 0x60); // 0|1|1|d9|d8|d7|d6|d5
    adcLO = (char)((adcValue & 0x1f) | 0xE0); // 1|1|1|d4|d3|d2|d1|d0

    putc(adcHI); // Transmit Byte 1 (d9...d5)
    putc(adcLO); // Transmit Byte 2 (d4...d0)

    /** Back to Channel 1 */
    set_adc_channel(0);
    delay_us(20); // Delay for sampling cap to charge
}

void SetBaudRate()
{
    switch(PORTE & 0x07) // Read dip switches and setup baud rate
    {
        case 0: set_uart_speed(4800); break;
        case 1: set_uart_speed(9600); break;
        case 2: set_uart_speed(14400); break;
        case 3: set_uart_speed(19200); break;
        case 4: set_uart_speed(32768); break;
    }
}

```

```

        case 5: set_uart_speed(38400); break;
        case 6: set_uart_speed(57600); break;
        case 7: set_uart_speed(115200); break;
    }
}

```

10.10. Mark 10.c (Test External RAM Chip)

This program carries out three tests: -

1. Fill all memory locations with 0xFFh and check.
2. Fill all memory locations with 0x00h and check.
3. Fill all memory locations with the LSB of the address and check.

RS232 communication is used to relay the results to the PC; the DOS based program test.exe used in RX terminal mode is used to receive and display the results. This program is designed to test the functions used to access the RAM (ReadRAM, WriteRAM), and every location on the RAM chip.

Notice that there are 1 cycle delays in the ReadRAM and WriteRAM functions these delays make sure that the address and data lines are value before a read or write operation is completed. Note the RAM chip used has a reassembly fast access time; hence the RAM may operate correctly with these delays removed. This program is in preparation for storage mode operation.

```

/* -----
| FILE      : mark_c10.c
| PROJECT   : Low Cost PC Based Oscilloscope
| DESC      : Test RAM Chip
| =====
| DATE      : 21/03/2002
| BY        : Colin K McCord
| VERSION   : 1.2
| ----- */

#include <16F877.h>
#define PIC16F877 * =16 ADC=10

// use #device adc = 10 to implement a 10-bit conversion,
// otherwise the default is 8-bits.

#fuses HS, NOWDT, NOPROTECT, NOLVP, NOBROWNOUT

#byte PORTB = 0x06 // PortB lives in File 6 (data bus)
#byte PORTC = 0x07 // PortC lives in File 7 (address bus A12 to A8, RX, TX & R/W)
#byte PORTD = 0x08 // PortD lives in File 8 (address bus A0 to A7)
#byte PORTE = 0x09 // PortE lives in File 9 (used for DIP switches)

#bit RW = PORTC.5 // RW = 0 for Write, and 1 for read

/* Note 20MHz clock must be used for 115,000 bps, the % error is to large at slower speeds */
#use delay(clock = 20000000)
#use rs232(baud=9600, xmit = PIN_C6, rcv = PIN_C7, parity = N, bits = 8)

#use fast_io(B) // Fast access to PortB (don't fiddle with TRISB)
#use fast_io(C) // Fast access to PortB (don't fiddle with TRISB)
#use fast_io(D) // Fast access to PortD (don't fiddle with TRISD)
#use fast_io(E) // Fast access to PortE (don't fiddle with TRISE)

/* Forward declaration of functions */
void SetBaudRate();
void WriteRAM(long int address, char data);
char ReadRAM(long int address);

main()
{
    long int i;
    long int pass, fail;
    set_tris_c(0x0C); // TRISD = 11000000; RC0 to RC5 TTL Outputs
                    // bits 7&6 must be set for UART to work
    set_tris_d(0x00); // TRISD = 00000000; RD0 to RD7 TTL Outputs
    port_b_pullups(FALSE); // Don't use internal pull up resistors
}

```

```
SetBaudRate();

while(TRUE)
{
    delay_ms(5000); // 5 Sec delay

    printf("TEST RAM CHIP...");

    /** RAM Test 1 write 0xFF into all locations and check **/
    printf("\n\nTest 1 - Fill RAM with 0xFF --->");
    pass = 0; // Reset Pass counter
    fail = 0; // Reset Fail counter

    for (i=0;i<8192;i++) // Write 0xFF to all locations
    {
        WriteRAM(i,0xFF);
    }

    for (i=0;i<8192;i++)
    {
        if(ReadRAM(i) == 0xFF) pass++;
        else fail++;
    }

    printf(" Passes = %lu, Fails = %lu", pass,fail);

    /** RAM Test 2 write 0x00 into all locations and check **/
    printf("\n\nTest 2 - Fill RAM with 0x00 --->");
    pass = 0; // Reset Pass counter
    fail = 0; // Reset Fail counter

    for (i=0;i<8192;i++) // Write 0x00 to all locations
    {
        WriteRAM(i,0x00);
    }

    for (i=0;i<8192;i++)
    {
        if(ReadRAM(i) == 0x00) pass++;
        else fail++;
    }

    printf(" Passes = %lu, Fails = %lu", pass,fail);

    /** RAM Test 3 write LSB of address into all locations and check **/
    printf("\n\nTest 3 - Fill RAM with LSB of Address --->");
    pass = 0; // Reset Pass counter
    fail = 0; // Reset Fail counter

    for (i=0;i<8192;i++) // Write LSB of address to all locations
    {
        WriteRAM(i,(char)i);
    }

    for (i=0;i<8192;i++)
    {
        if(ReadRAM(i) == (char)i) pass++;
        else fail++;
    }

    printf(" Passes = %lu, Fails = %lu", pass,fail);

    printf("\n\nEnd of RAMTEST, Hopefully work can now start on storage mode.\n\n");

}

sleep();
}
```

```
void SetBaudRate()
{
    // Verison 1.1, 22/3/2002, by CKM

    set_tris_e(0x17);    // TRISE = 00010111; RE2,RE1 and RE0 Inputs
                        // Parallel slave mode is ON, e.g. TTL Inputs, on Port E and D

    switch(PORTE & 0x07) // Read dip switches and setup baud rate
    {
        case 0: set_uart_speed(4800); break;
        case 1: set_uart_speed(9600); break;
        case 2: set_uart_speed(14400); break;
        case 3: set_uart_speed(19200); break;
        case 4: set_uart_speed(32768); break;
        case 5: set_uart_speed(38400); break;
        case 6: set_uart_speed(57600); break;
        case 7: set_uart_speed(115200); break;
    }

    set_tris_e(0x07); // Turn OFF Parallel Slave Mode as this affects port D (Address Bus)
}

void WriteRAM(long int address, char data)
{
    // 20/03/2002, version 1.0 by Colin McCord

    set_tris_b(0x00);    // TRISE = 00000000; RB0 to RB7 TTL Outputs (data bus)

    RW = 1;              // Don't Write until address and data bus is setup.

    // Set Address
    PORTD = (char)address; // Set A7..A0
    PORTC = ((PORTC & 0xE0) | (address>>8)); // Set A12..A8

    PORTB = data;        // Put data on the data bus

    delay_cycles(1);     // 200nS delay, same as NOP, (1/20,000,000)*4*1) = 200nS
                        // make sure address & data is valid before write

    RW = 0;              // Write data to specified address

    delay_cycles(1);     // 200nS delay (same as NOP), make sure write is finished
    RW = 1;              // Back to Read Mode
}

char ReadRAM(long int address)
{
    // 20/03/2002, version 1.0 by Colin McCord

    set_tris_b(0xFF);    // TRISE = 11111111; RB0 to RB7 TTL Inputs

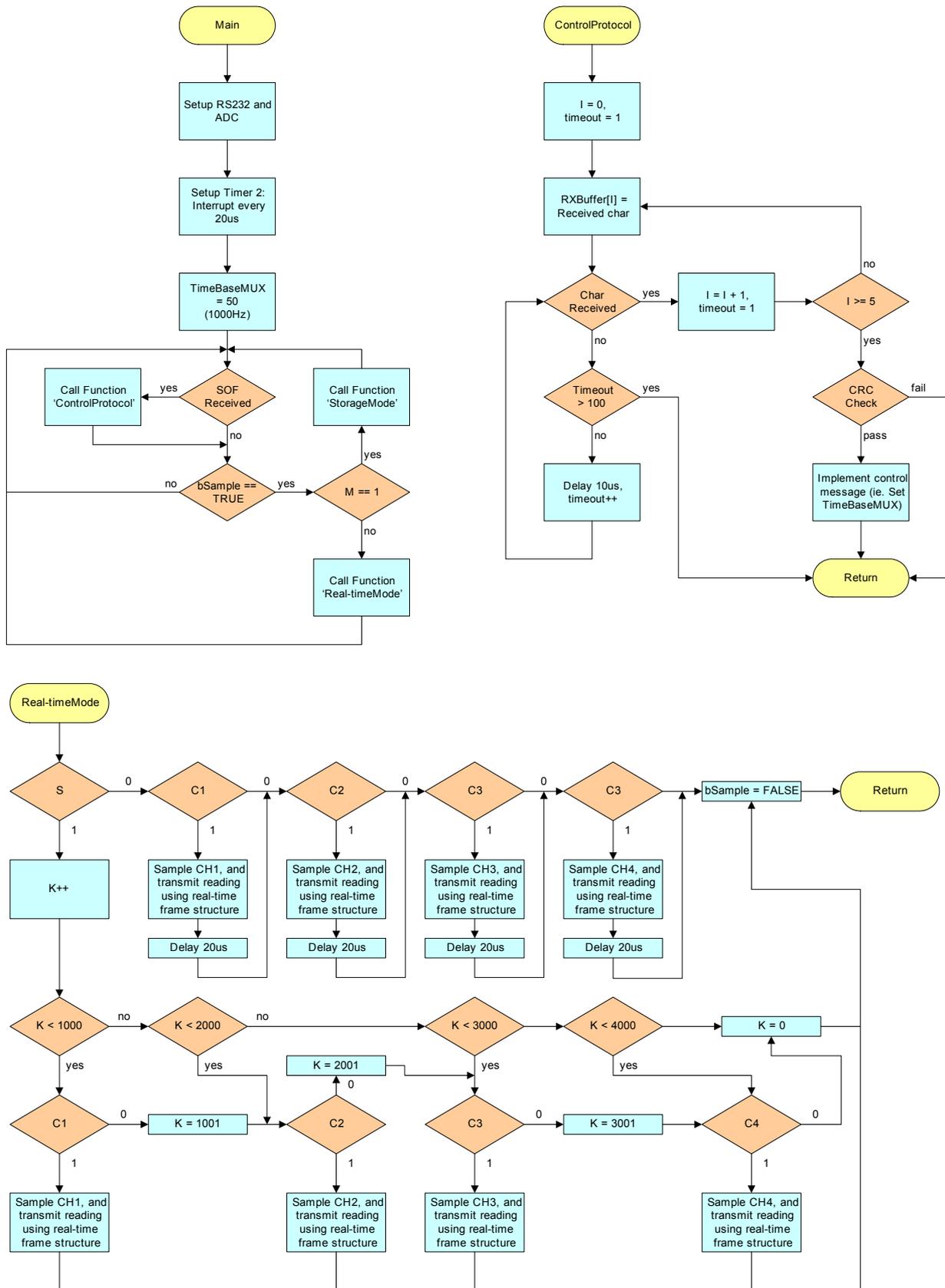
    RW = 1;              // Put RAM chip in Read Mode

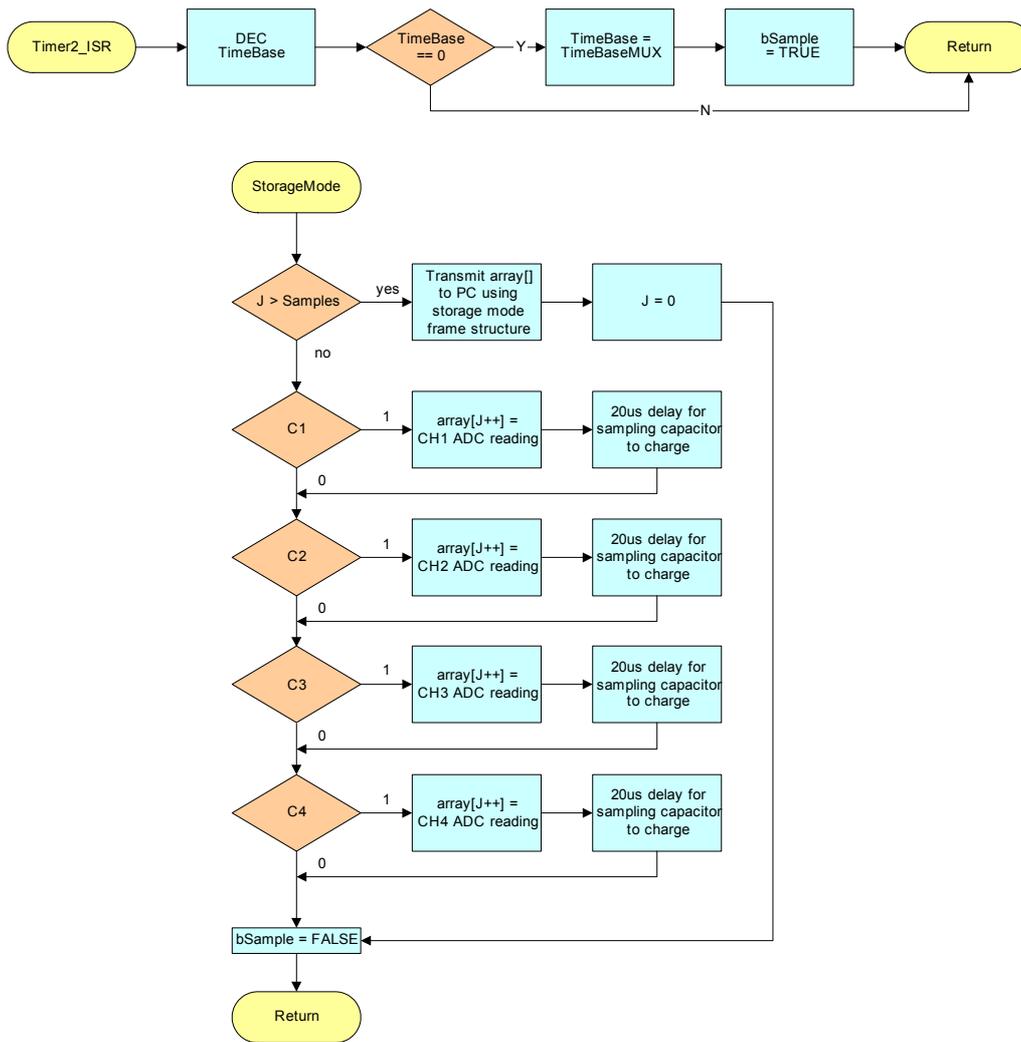
    // Set Address
    PORTD = ((char)address); // Set A7..A0
    PORTC = ((PORTC & 0xE0) | (address>>8)); // Set A12..A8

    delay_cycles(1);     // 200nS delay, same as NOP, (1/20,000,000)*4*1) = 200nS
                        // make sure data is valid before reading

    return PORTB;
}
```

10.11. Mark 11.c (Main Program)





Note the Mark11.c program is not complete and the flowcharts above are draft flowcharts.

Clearly the test was a success, and it has been proven that the transmit mode of the RS232 communications is working. Note: Baud rate = 9600bps, OSC = 4 MHz, RS232 cable length (unshielded) = 1.5m.

11.1.2. Test 2: Fully Test RS232 Communications

Program mark2.c (Version: 1.0) was compiled and loaded into PIC using Microchip MPLAB. This program is extremely simple; basically it waits for an incoming character and transmits it. This is a very simple method of fully testing the RS232 communication (PIC software, PIC, MAX232, cable, and wiring). The DOS based test program (test.exe, version 1.05) was used to carryout two separate tests. The first used the TX terminal to transmit a message, then the RX terminal is used to receive the message, if the RX & TX messages are identical the test is successful. The second used the loop back test function of the test program; this stresses the communications as a large amount of data is transmitted and compared with the received block of data.

Screen dump of test program (test.exe) in TX terminal mode then RX terminal mode: -

```
TX Terminal
-----
Type your message, hit [ENTER] to transmit: -

Testing RS232 Communications

Sending Messaging: -
Testing RS232 Communications

Message sent successfully.

RX Terminal
-----
Waiting for message (press a key to stop)
Testing RS232 Communications
```

Screen dump of test program (test.exe) after loop back test was completed: -

```
Test Started: 13:37:31
-----
Testing          .... 100% - PASS
* Test 1 - PASS
* Test 2 - PASS
* Test 3 - PASS
-----
Test Finished: 13:37:33

Hit any [Key] to return to main menu
```

Clearly the RS232 communication tests were successful, note: baud rate = 9600bps, OSC = 4 MHz, RS232 cable length (unshielded) = 1.5m.

11.1.3. Test 3: Test ADC

Program mark3.c (Version: 1.1) was compiled and loaded into PIC using Microchip MPLAB. This program is a little more complex than the first two, basically it reads ADC CH1 (AN0), and transmits the result through RS232 using the real-time frame format, every 10ms (sample rate of 100Hz). The scope.exe (version 1.023a) program was used to display the real-time frames graphically, and a calibrated digital signal generator was use to generate a 0 to 5V (e.g. 5V peak to peak with a DC offset of 2.5V) sinewave at a variable frequency (e.g. 2Hz, 5Hz, 10Hz, etc...). Portable commercial digital storage oscilloscope was used to make sure that the signal generator output was 0 to 5V as values outside this range would damage the PICs ADC (analogue circuit not constructed at this stage).

Note: Sample rate = 100Hz, 100 horizontal samples are displayed on the scope display (10 samples per division). Hence scope time-base is 0.1 seconds per division that's 10Hz. Note: time-base is fixed on this version of the scope program.

1Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.1.3a. 10 division per cycle, hence the calculated frequency of the waveform is $1/(10 * 0.1) = 1\text{Hz}$.

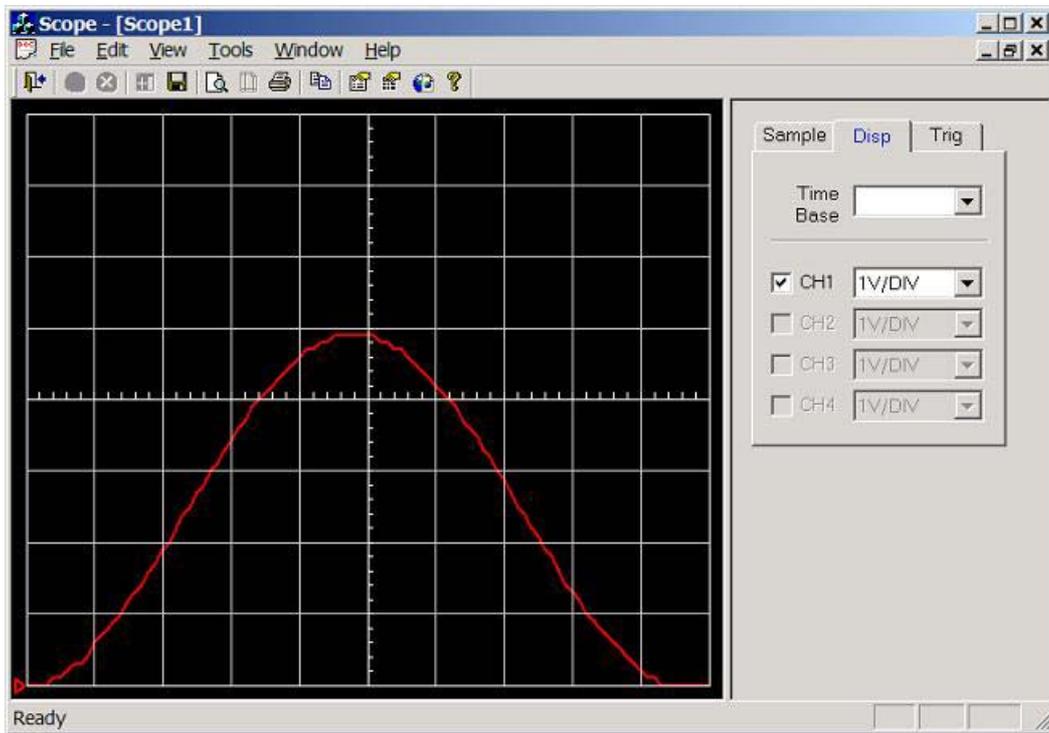


Figure 11.1.3a. Screen dump of scope program (sample rate = 100Hz, input wave = 1Hz sinewave 0-5V)

2Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.1.3b. 5 divisions per cycle, hence the calculated frequency of this waveform is $1/(5 * 0.1) = 2\text{Hz}$.

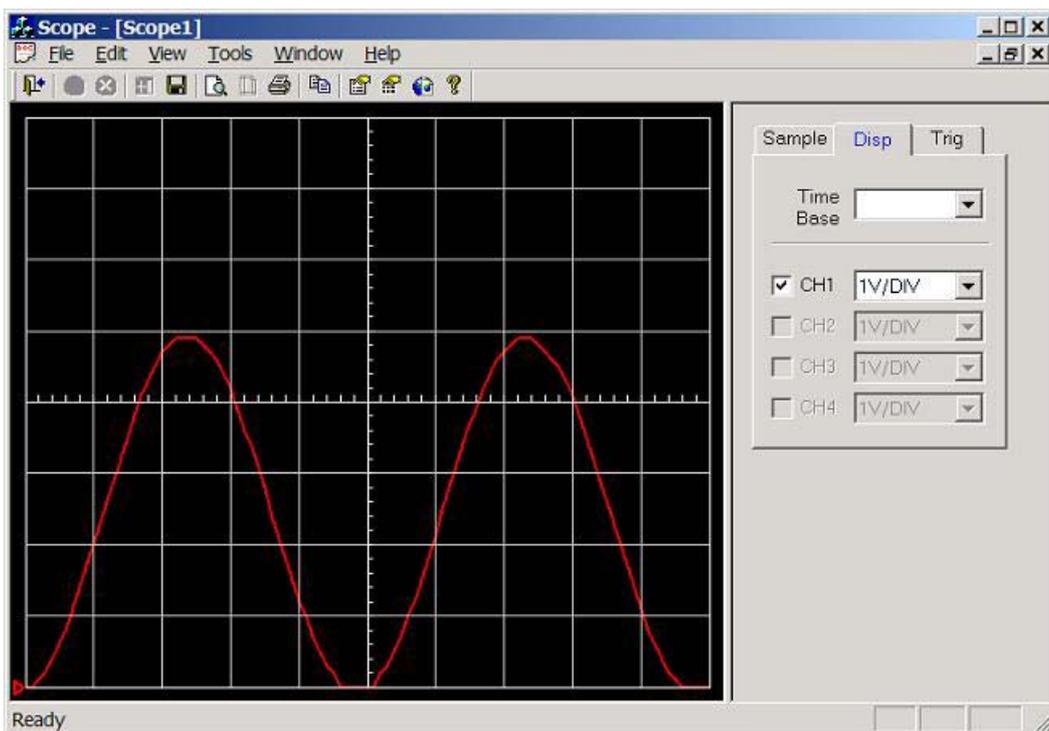


Figure 11.1.3b. Screen dump of scope program (sample rate = 100Hz, input wave = 2Hz sinewave 0-5V)

5Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.1.3c. 2 divisions per cycle, Hence the calculated frequency of this waveform is $1/(2 * 0.1) = 5\text{Hz}$.

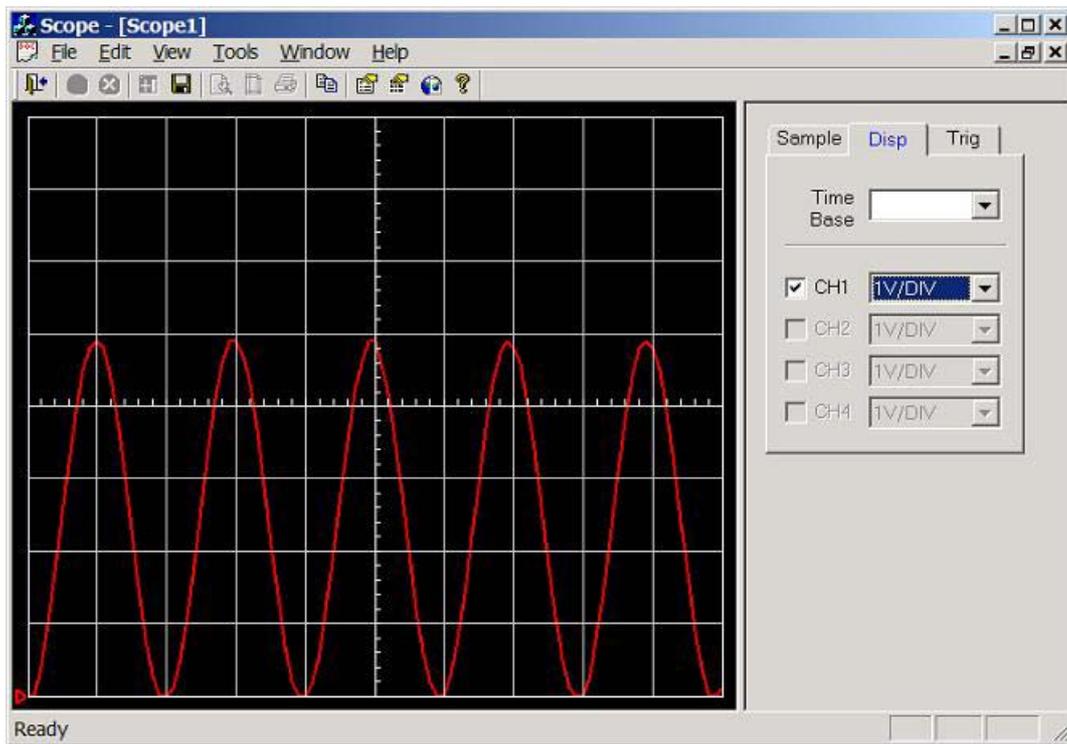


Figure 11.1.3c. Screen dump of scope program (sample rate = 100Hz, input wave = 5Hz sinewave 0-5V)

10Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.1.3d. 1 divisions per cycle, hence the calculated frequency of this waveform is $1/(1 * 0.1) = 10\text{Hz}$.

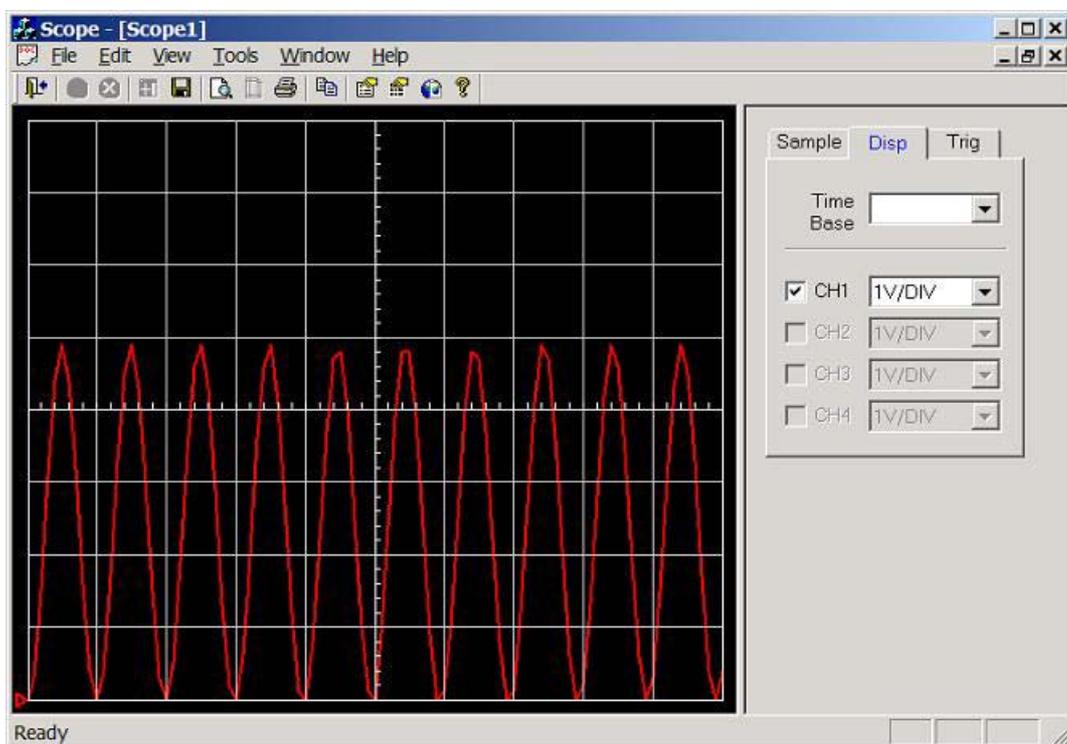


Figure 11.1.3d. Screen dump of scope program (sample rate = 100Hz, input wave = 10Hz sinewave 0-5V)

20Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.1.3e. 0.5 divisions per cycle, hence the calculated frequency of this waveform is $1/(0.5 * 0.1) = 20\text{Hz}$. But waveform is under sampled and heavily

distorted, recall that the pulse interpolator display type (scope.exe uses this) required at least 10 sample / cycle before a recognisable sinewave can be plotted.

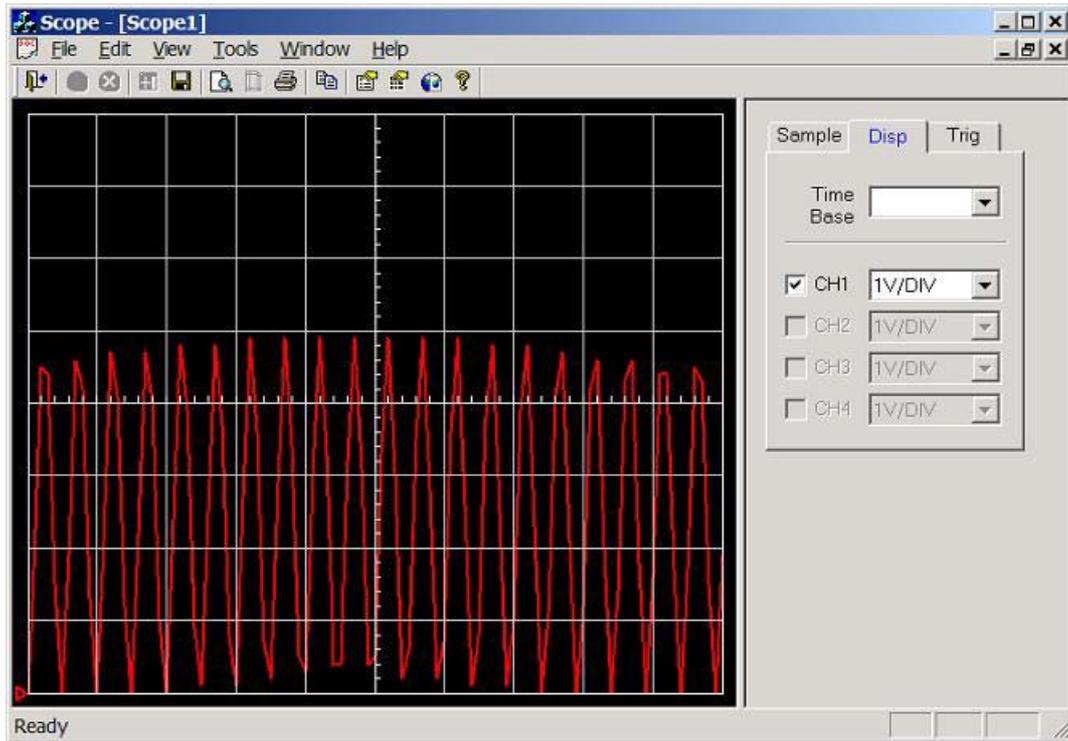


Figure 11.1.3e. Screen dump of scope program (sample rate = 100Hz, input wave = 20Hz sinewave 0-5V)

50Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.1.3f. Something real strange has happened here, clearly the result of under sampling.

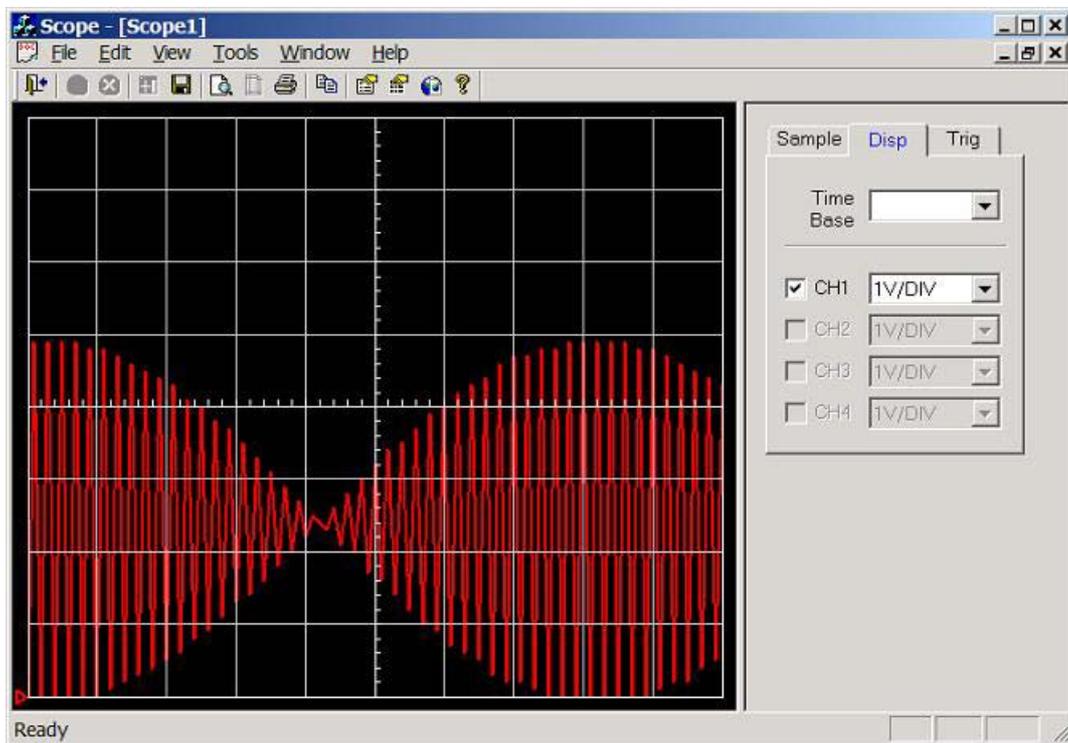


Figure 11.1.3f. Screen dump of scope program (sample rate = 100Hz, input wave = 50Hz sinewave 0-5V)

100Hz sine-wave (0-5V) connected to CH1 (AN0), see figure 11.1.3g. 10 divisions per cycle, hence the calculated frequency of this waveform is $1/(10 * 0.1) = 1\text{Hz}$. Clearly not correct, the reason for this is that the

sample rate is not high enough and because sine-waves are periodical it is possible to get false time-base readings (e.g. aliasing). This will be a mayor concern at a later stage in the project, and will be investigated fully.

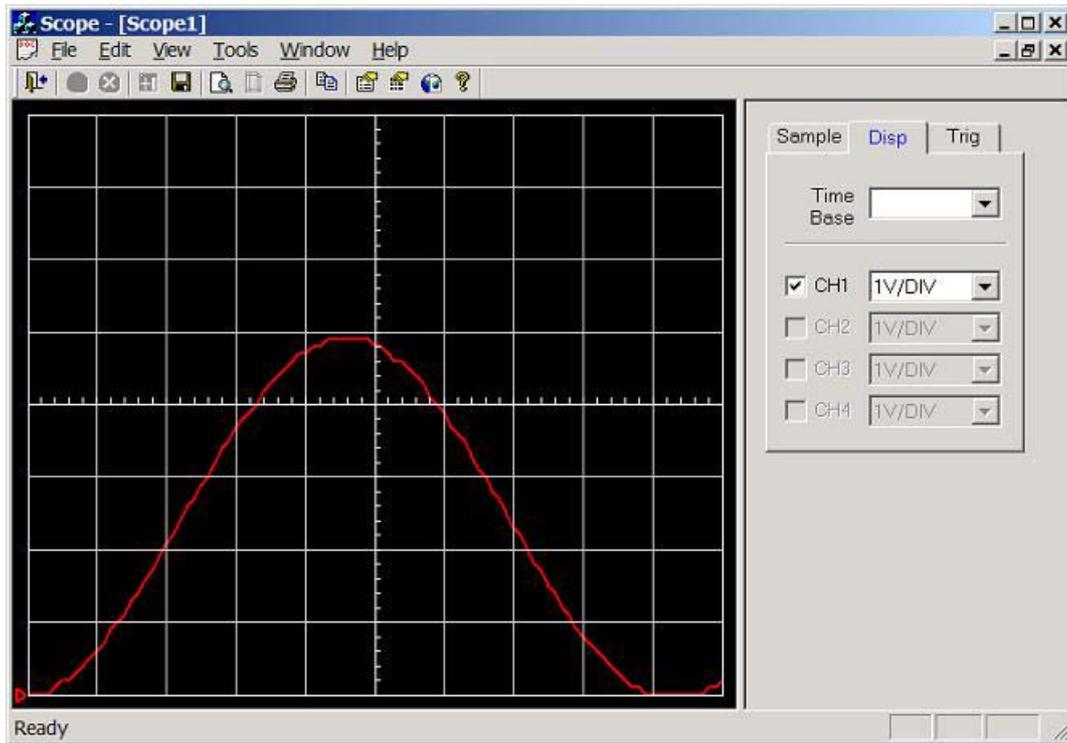


Figure 11.1.3g. Screen dump of scope program (sample rate = 100Hz, input wave = 100Hz sinewave 0-5V)

200Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.1.3h. 5 divisions per cycle, hence the calculated frequency of this waveform is $1/(5 * 0.1) = 2\text{Hz}$. This signal is clearly an alias of the original signal (false signal).

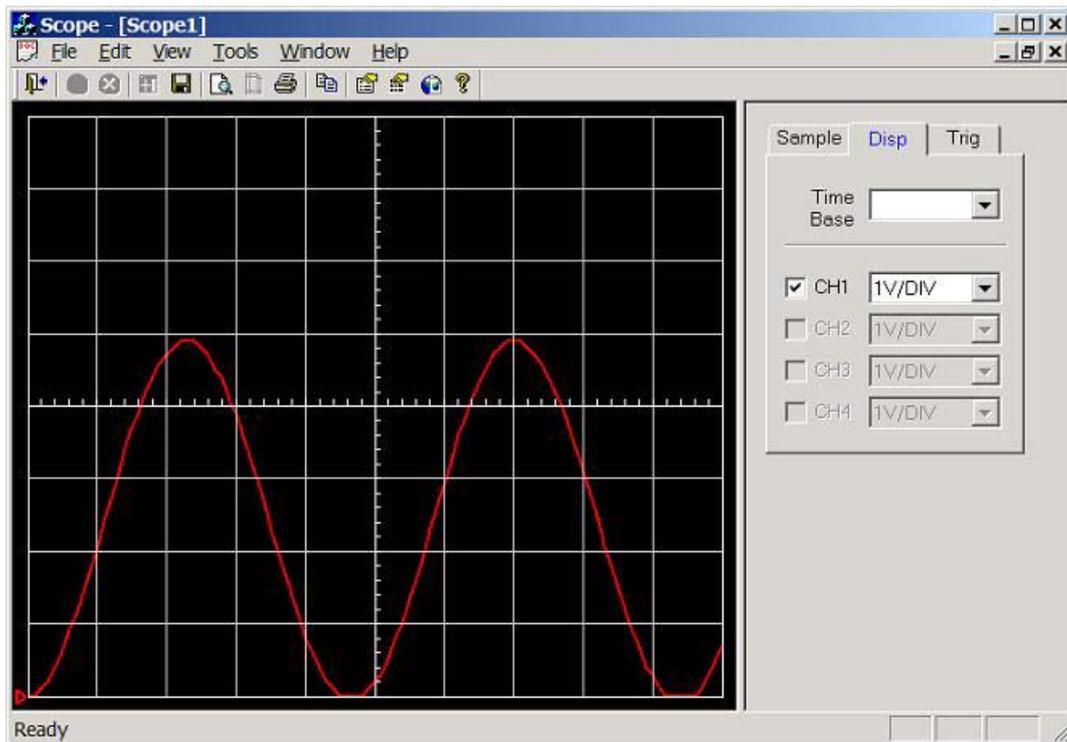


Figure 11.1.3h. Screen dump of scope program (sample rate = 100Hz, input wave = 200Hz sinewave 0-5V)

11.1.4. Conclusion

Clearly the three tests that were carried out were completely successful and work can now start on the main real-time embedded program. Test 3, was extremely useful, as it was able to test fully the graphical display for a fixed sample rate. The triggering (negative to positive, 0 to positive in this case) worked, but the waveform was not completely stable, as the amplitude of samples varied slightly (due to noise, or/and tolerance of ADC) hence the waveform was continuously changing very slightly; sampling when the PIC is sleeping may reduce noise, hence a more stable waveform could be displayed (too slow to be practical). The PIC datasheet recommends not to use the ADCs internal oscillator when the PIC is running above 1MHz (unless sampling is done when PIC is sleeping), mark3.c uses the ADCs internal oscillator (not good) and this could explain why the waveform was continuously changing slightly (e.g. noise from PIC oscillator).

It was discovered that it is possible for the scope program to display perfect sine-waves at false time-based readings when the waveform is under sampled. This is a common problem which all digital scopes have called 'aliasing', frequency above half the sample frequency should be filter out. Note it may be possible to use this as an advantage as it makes it possible to monitor waveforms much higher than the sampling rate, improving the performance of the PC based scope program (if the real frequency could be detected).

11.2. Test Session 2: FG Wilson Lab (05/03/2002)

The circuit diagram shown in figure 11.2a was constructed on prototype board, with RS232 communication connected to laptop (COM1, 1GHz AMD processor, 256MHz SDRAM, 8MB 3D accelerator card, running windows XP).

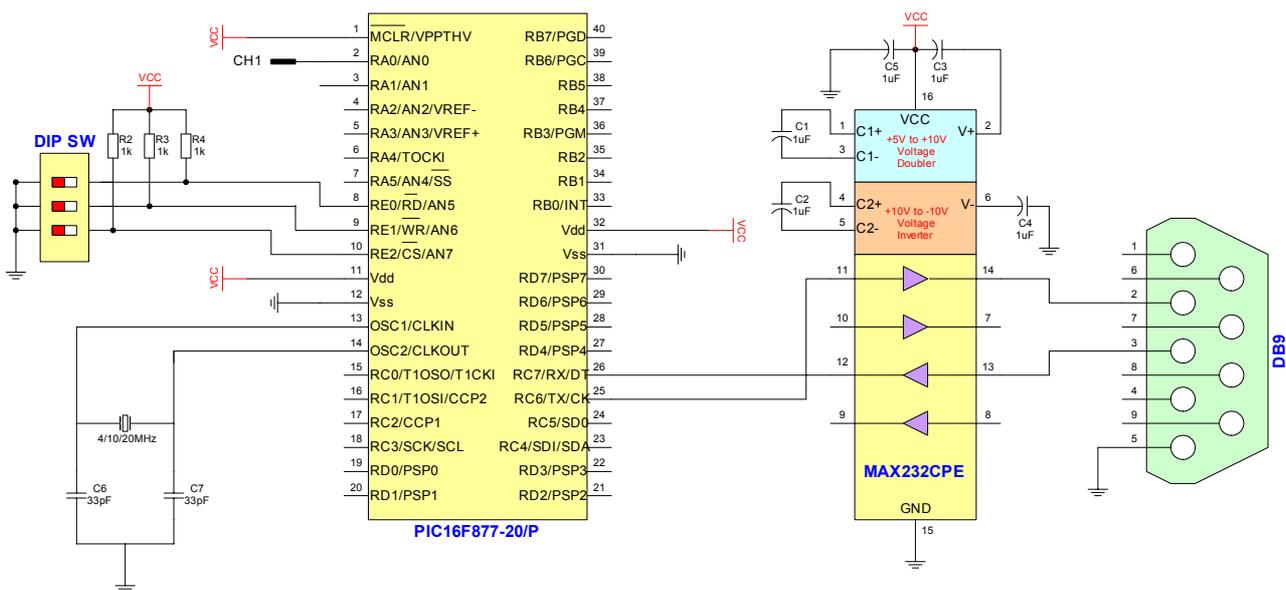


Figure 11.2a. Test Circuit Diagram 2

11.2.1. Test 4: Test RS232 Communications (Baud Rate set by DIP Switches)

Program mark4.c was compiled and loaded into the PIC using Microchip MPLAB. This program is extremely simple (basically the same as mark1.c expect baud rate is set by the DIP switches), basically it reads the DIP switches and sets the baud rate, then it transmits "Testing..." continuously with a 1 second delay between each "Testing...". The DOS based test program (test.exe, version 1.05) was used in the RX terminal mode to receive incoming characters.

All baud rates were tested and there were no problems.

11.2.2. Text 5: Fully Test RS232 Communications (Baud Rate set by DIP Switches)

Program mark5.c was compiled and loaded into PIC using Microchip MPLAB. This program is extremely simple; basically it reads the DIP switches and sets the baud rate, then it waits for an incoming character and transmits it. This is a very simple method of fully testing the RS232 communication (PIC software, PIC, MAX232, cable, and wiring). The DOS based test program (test.exe, version 1.05) was used to carry out two separate tests. The first used the TX terminal to transmit a message, then the RX terminal is used to receive the message, if the RX & TX messages are identical the test is successful. The second used the loop back test function of the test program; this stresses the communications as a large amount of data is transmitted and compared with the received block of data.

The TX/RX terminal tests were successful for all baud rates, but the loop back tests failed at high baud rates. There are two possible reasons for this; the first is that because a large block of data is being transmitted through unshielded cable, clearly there is a good chance of an error occurring at high baud rates. The second possibility is that the PIC UART input buffer overflowed.

11.2.3. Test 6: Test ADC, Scope Program and Real-time Communication Protocol

Program mark6.c was compiled and loaded into PIC using Microchip MPLAB, basically it reads the DIP switches and sets the baud rate, then it reads the ADC CH1 (AN0), and transmits the result through RS232 using the real-time frame format, every 1ms (sample rate of 1000Hz). The scope.exe (version 1.027a) program was used to display the real-time frames graphically, and a calibrated digital signal generator was used to generate a 0 to 5V (e.g. 5V peak to peak with a DC offset of 2.5V) sine wave at a variable frequency (e.g. 2Hz, 5Hz, 10Hz, etc...).

Note: Sample rate = 1000Hz, 10 horizontal samples per division. Hence scope time-base is 0.01 seconds per division that's 100Hz, note: time-base is fixed on this version of the scope program and was set to 115Kbaud.

2.5Hz sine wave (0-5V) connected to CH1/AN0, see figure 11.2.3a (high resolution 800 x 160, that's 80 x 16 divisions), 40 division per cycle, hence the calculated frequency of the waveform is $1/(40 * 0.01) = 2.5\text{Hz}$.

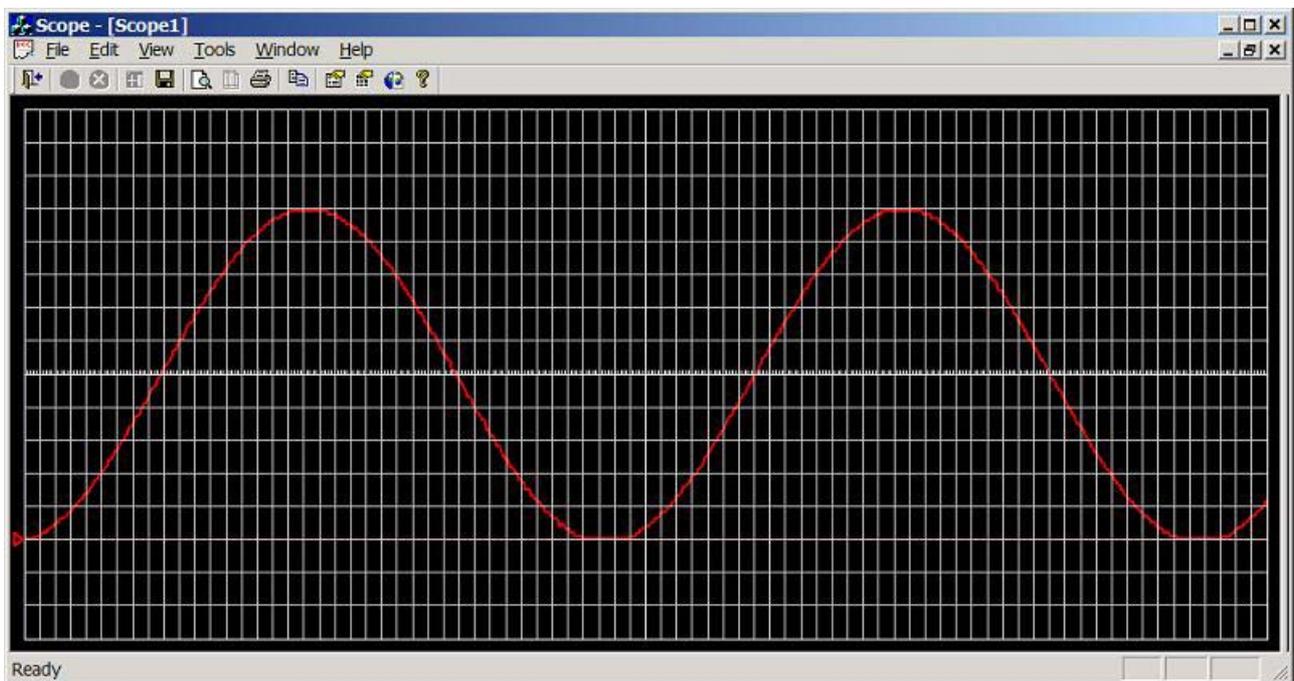


Figure 11.2.3a. Screen dump of scope program (sample rate = 1,000Hz, input wave = 2.5Hz sine wave 0-5V)

5Hz sine wave (0-5V) connected to CH1/AN0, see figure 11.2.3b (resolution 200 x 160, that's 20 x 16 divisions), 19 division per cycle, hence the calculated frequency of the waveform is $1/(19 * 0.01) = 5.26\text{Hz}$.

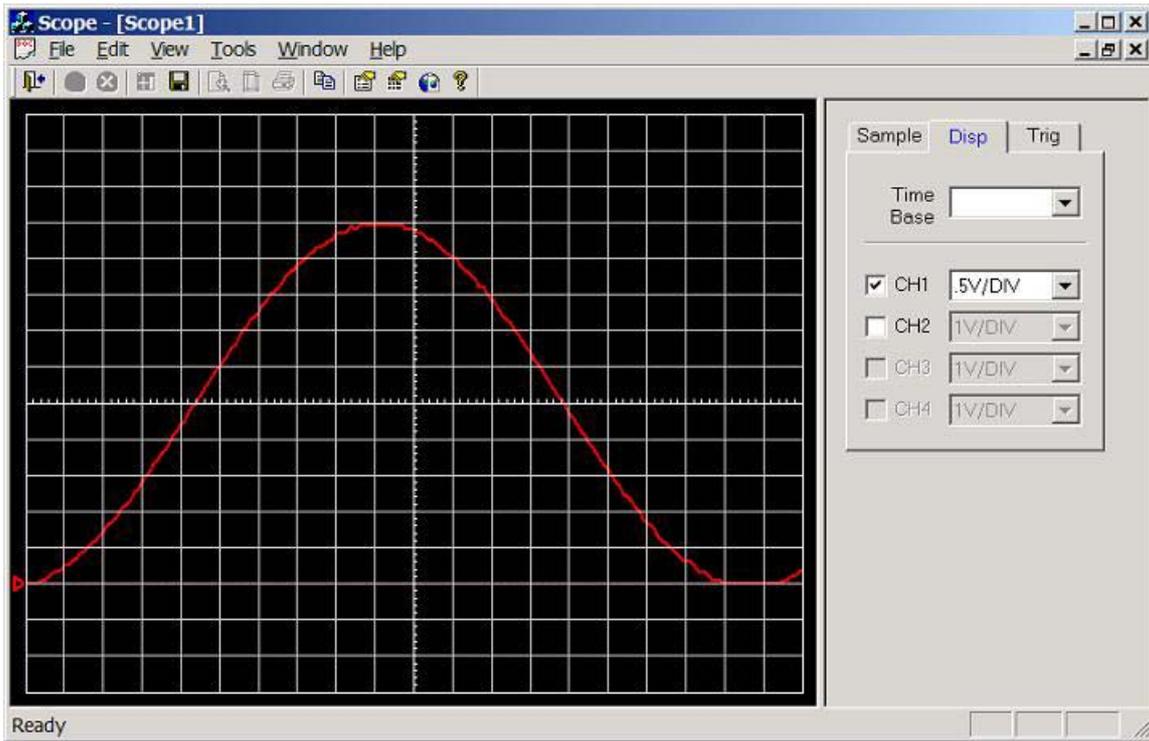


Figure 11.2.3b. Screen dump of scope program (sample rate = 1,000Hz, input wave = 5Hz sinewave 0-5V)

10Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3c (resolution 200 x 160, that's 20 x 16 divisions), 10 division per cycle, hence the calculated frequency of the waveform is $1/(10 * 0.01) = 10\text{Hz}$.

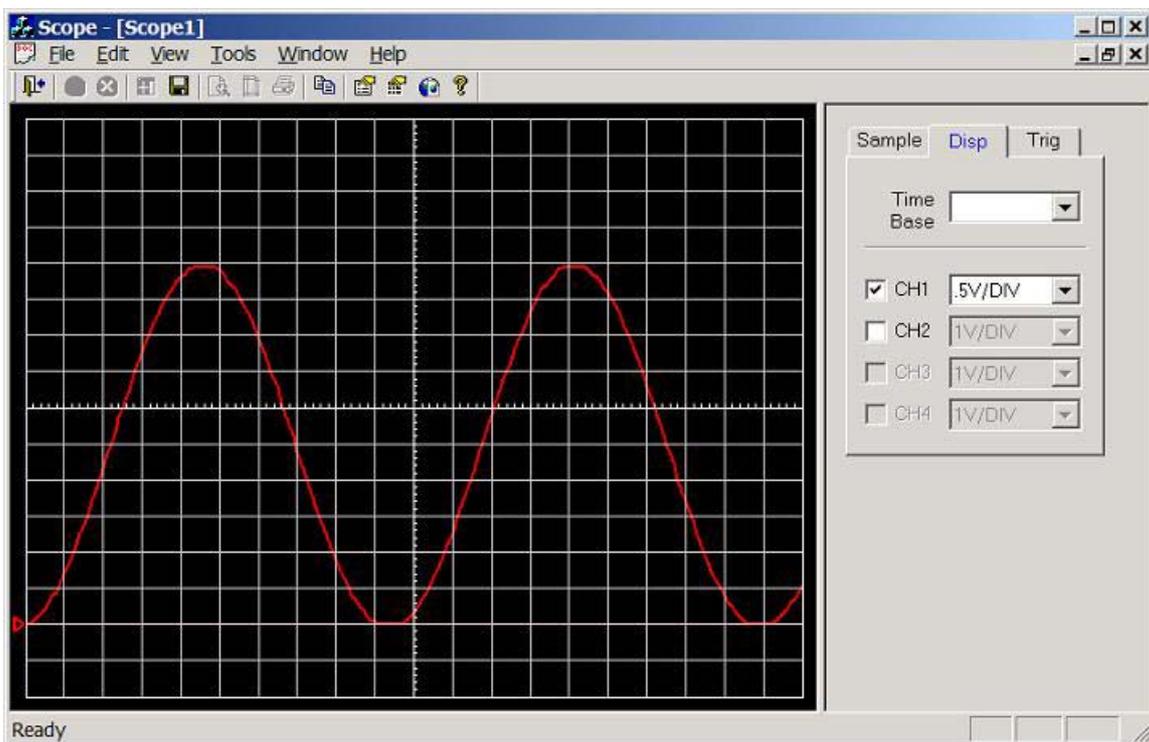


Figure 11.2.3c. Screen dump of scope program (sample rate = 1,000Hz, input wave = 10Hz sinewave 0-5V)

20Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3d (resolution 200 x 160, that's 20 x 16 divisions), 5 division per cycle, hence the calculated frequency of the waveform is $1/(5 * 0.01) = 20\text{Hz}$.

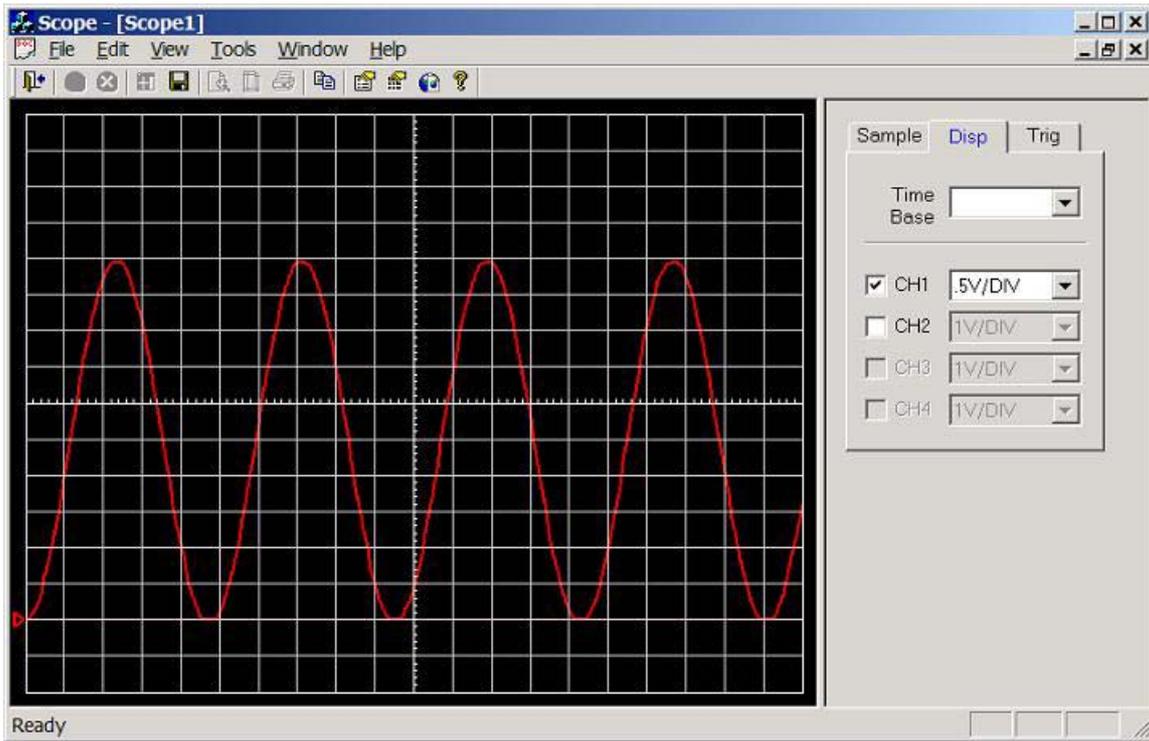


Figure 11.2.3d. Screen dump of scope program (sample rate = 1,000Hz, input wave = 20Hz sinewave 0-5V)

50Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3e (resolution 200 x 160, that's 20 x 16 divisions), 2 division per cycle, hence the calculated frequency of the waveform is $1/(2 * 0.01) = 50\text{Hz}$.

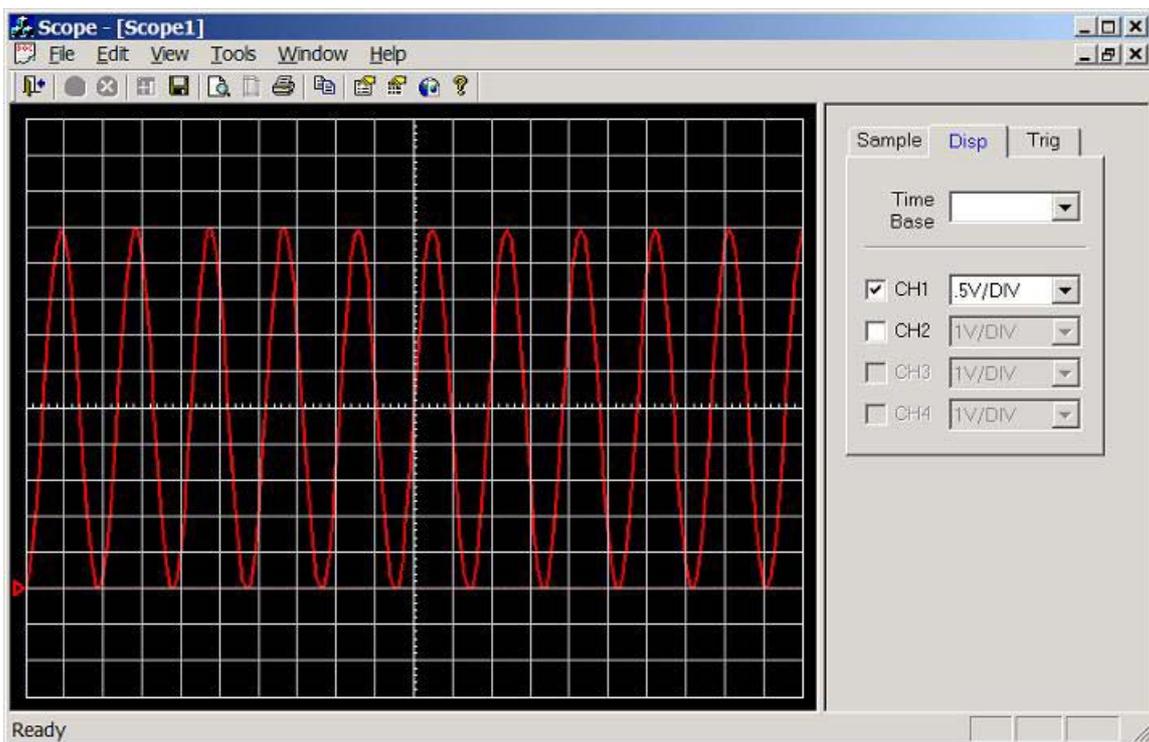


Figure 11.2.3e. Screen dump of scope program (sample rate = 1,000Hz, input wave = 50Hz sinewave 0-5V)

100Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3f (resolution 100 x 80, that's 10 x 8 divisions), 1 division per cycle, hence the calculated frequency of the waveform is $1/(1 * 0.01) = 100\text{Hz}$.

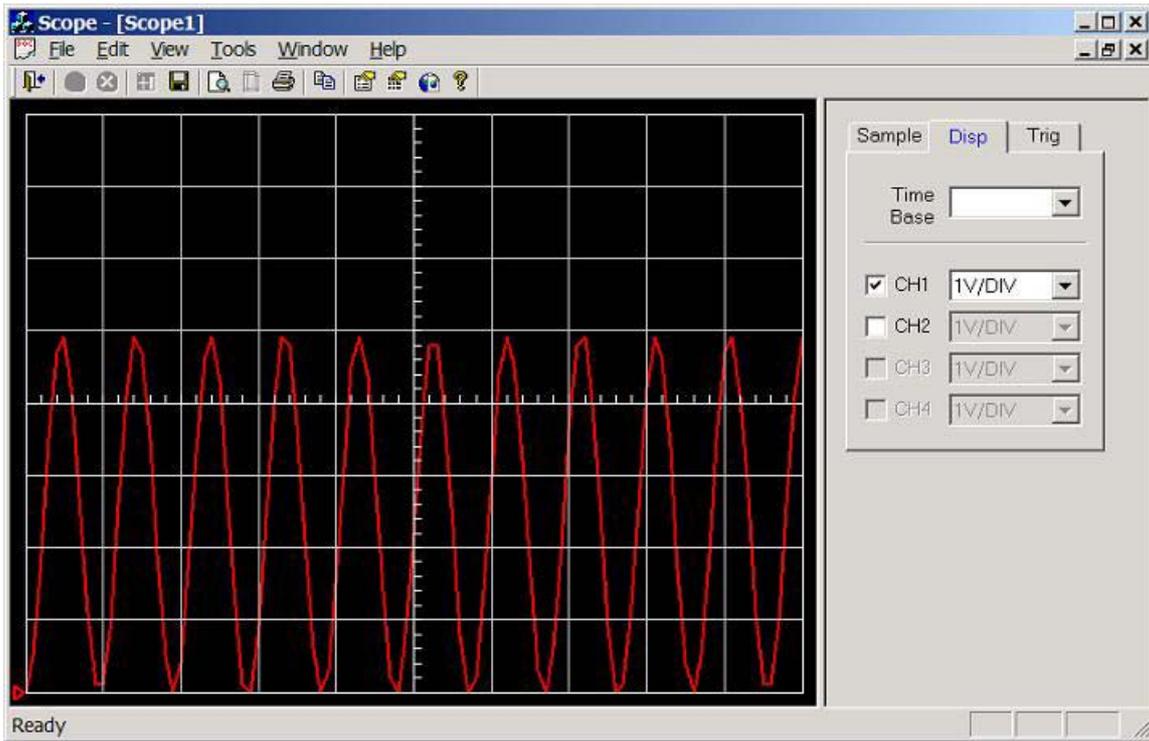


Figure 11.2.3f. Screen dump of scope program (sample rate = 1,000Hz, input wave = 100Hz sine wave 0-5V)

200Hz sine wave (0-5V) connected to CH1/AN0, see figure 11.2.3g (resolution 100 x 80, that's 10 x 8 divisions), 0.5 divisions per cycle, hence the calculated frequency of this waveform is $1/(0.5 * 0.01) = 200\text{Hz}$. But waveform is under sampled and heavily distorted, recall that the pulse interpolator display type (scope.exe uses this) requires at least 10 sample / cycle before a recognisable sine-wave can be plotted.

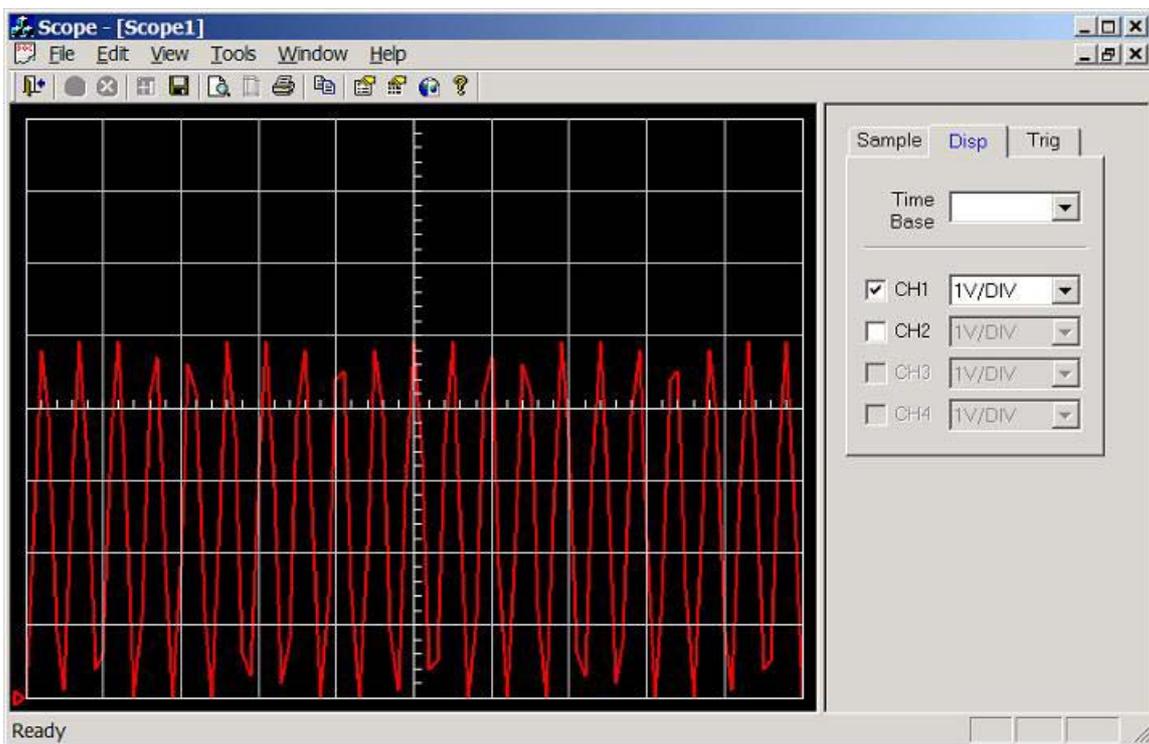


Figure 11.2.3g. Screen dump of scope program (sample rate = 1,000Hz, input wave = 200Hz sine wave 0-5V)

500Hz sine-wave (0-5V) connected to CH1/AN0, see figure 11.2.3h (resolution 100 x 80, that's 10 x 8 divisions), something real strange has happened here, clearly the result of under sampling.

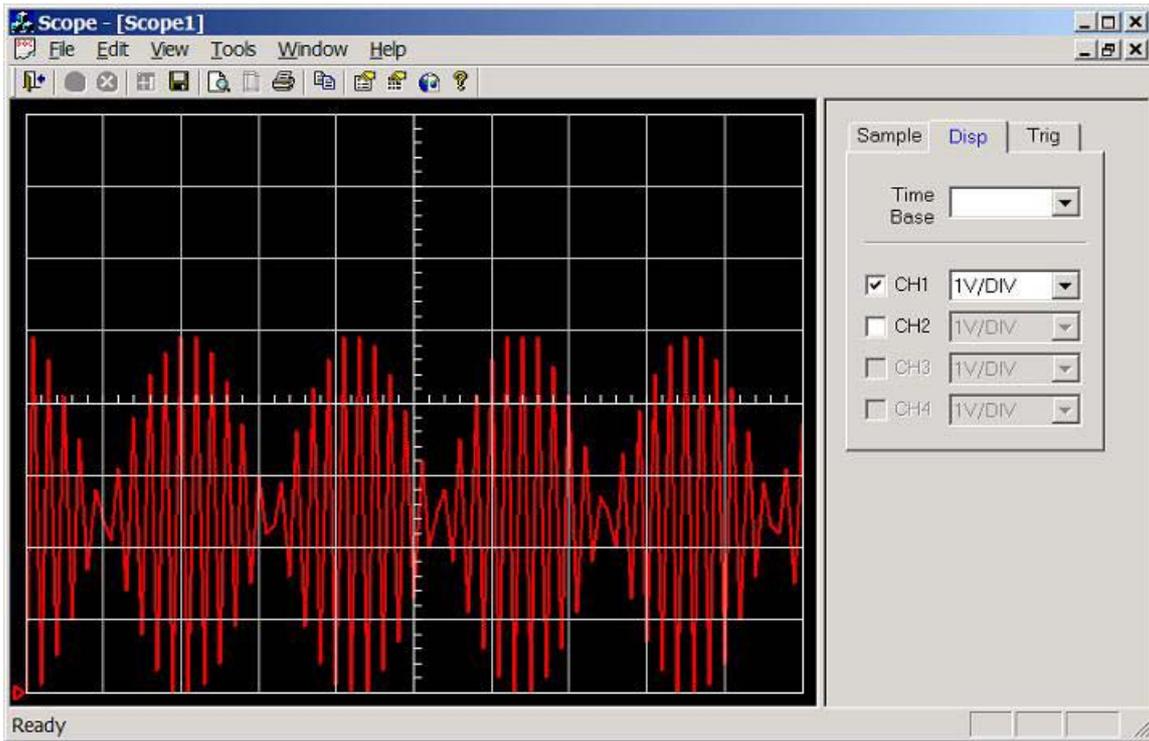


Figure 11.2.3h. Screen dump of scope program (sample rate = 1,000Hz, input wave = 500Hz sinewave 0-5V)

11.2.3.1. Sample Rate Changed to 2000Hz (mark6b.c)

Scope time-base is 0.005 seconds per division that's 200Hz.

10Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3.1a (resolution 200 x 160, that's 20 x 16 divisions), 19.5 division per cycle, hence the calculated frequency of the waveform is $1/(19.5 * 0.005) = 10.25\text{Hz}$.

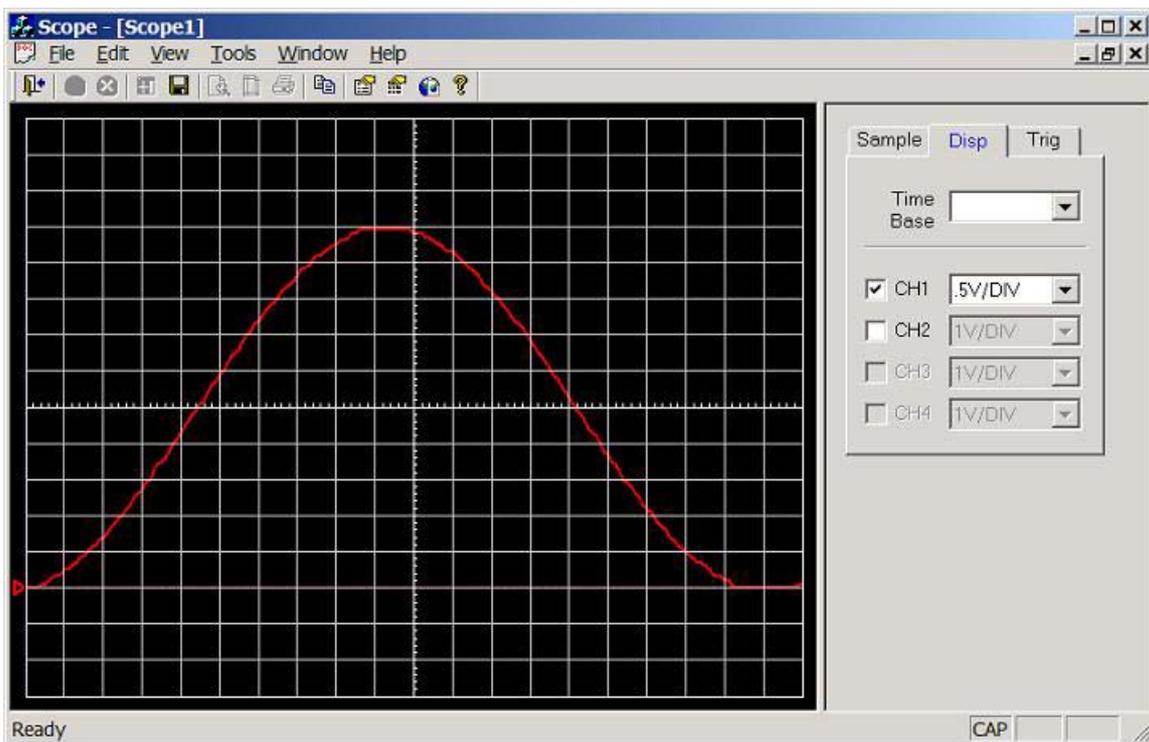


Figure 11.2.3.1a. Screen dump of scope program (sample rate = 2,000Hz, input wave = 10Hz sinewave 0-5V)

20Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3.1b (resolution 200 x 160, that's 20 x 16 divisions), 9.7 division per cycle, hence the calculated frequency of the waveform is $1/(9.7 * 0.005) = 20.6\text{Hz}$.

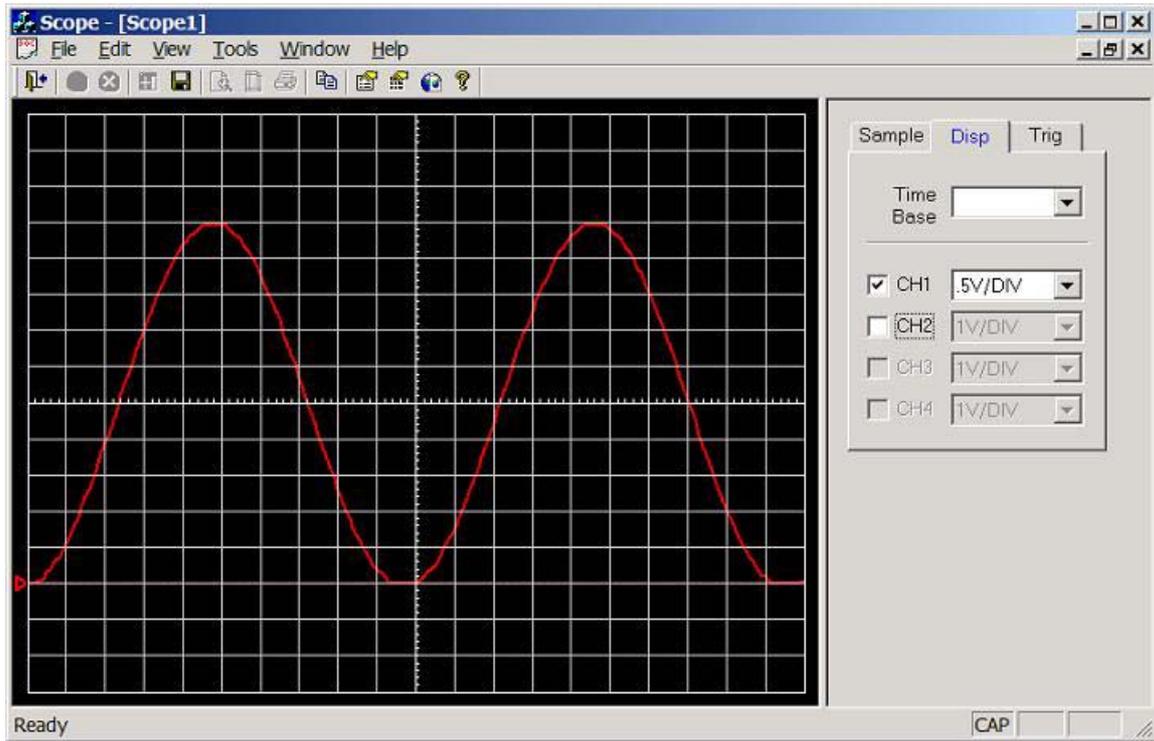


Figure 11.2.3.1b. Screen dump of scope program (sample rate = 2,000Hz, input wave = 20Hz sinewave 0-5V)

50Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3.1c (resolution 200 x 160, that's 20 x 16 divisions), 3.9 division per cycle, hence the calculated frequency of the waveform is $1/(3.9 * 0.005) = 51.3\text{Hz}$.

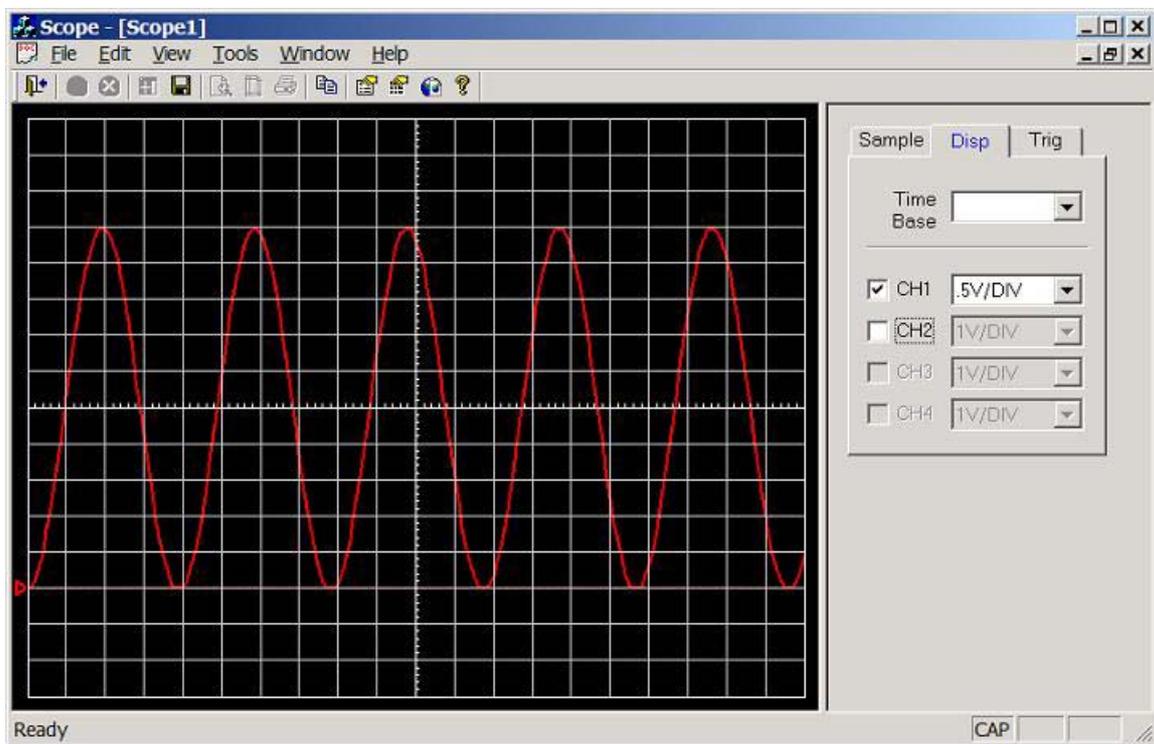


Figure 11.2.3.1c. Screen dump of scope program (sample rate = 2,000Hz, input wave = 50Hz sinewave 0-5V)

100Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3.1d (resolution 200 x 160, that's 20 x 16 divisions), 1.95 division per cycle, hence the calculated frequency of the waveform is $1/(1.95 * 0.005) = 102.5\text{Hz}$.

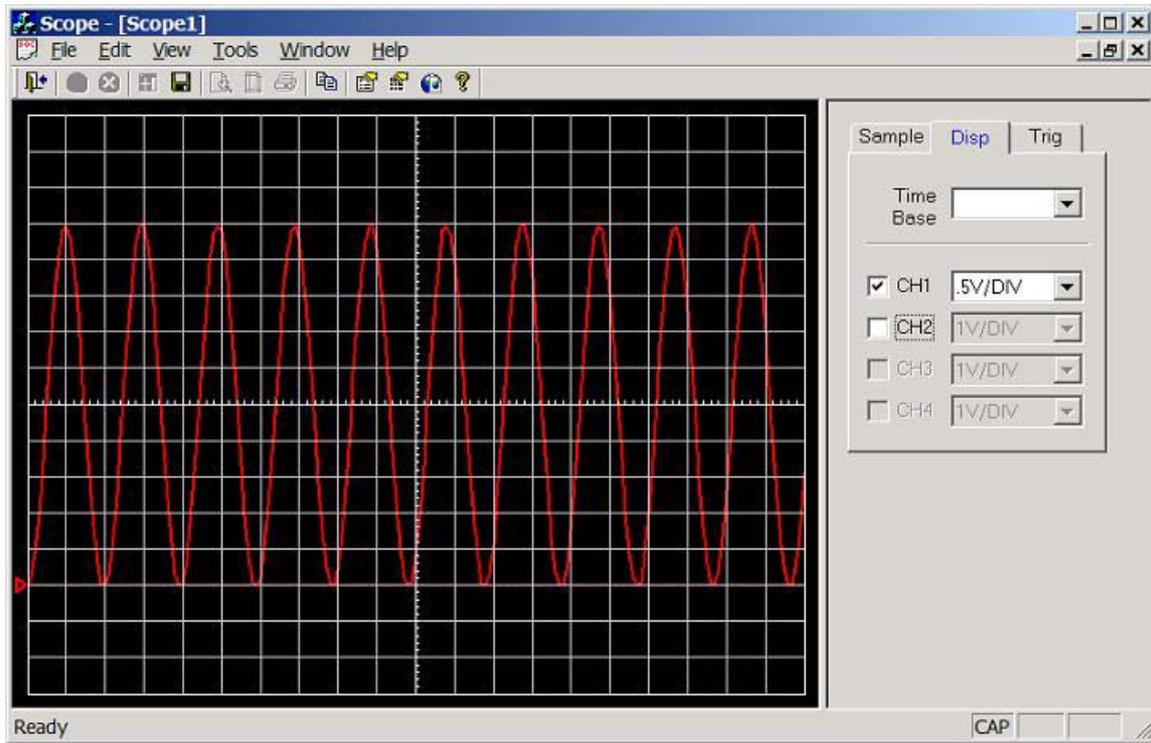


Figure 11.2.3.1d. Screen dump of scope program (sample rate = 2,000Hz, input wave = 100Hz sinewave 0-5V)

200Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3.1e (resolution 100 x 80, that's 10 x 8 divisions), 0.975 division per cycle, hence the calculated frequency of the waveform is $1/(0.975 * 0.005) = 205\text{Hz}$.

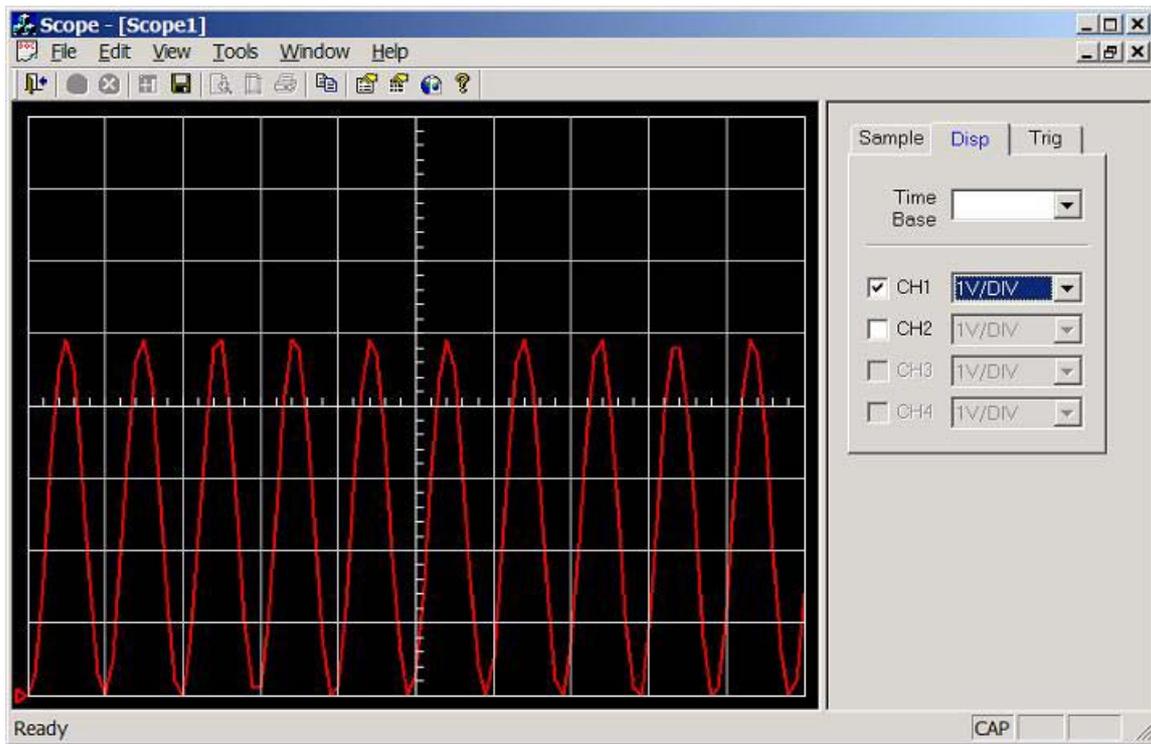


Figure 11.2.3.1f. Screen dump of scope program (sample rate = 2,000Hz, input wave = 200Hz sinewave 0-5V)

500Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3.1g (resolution 100 x 80, that's 10 x 8 divisions), clearly under sampled.

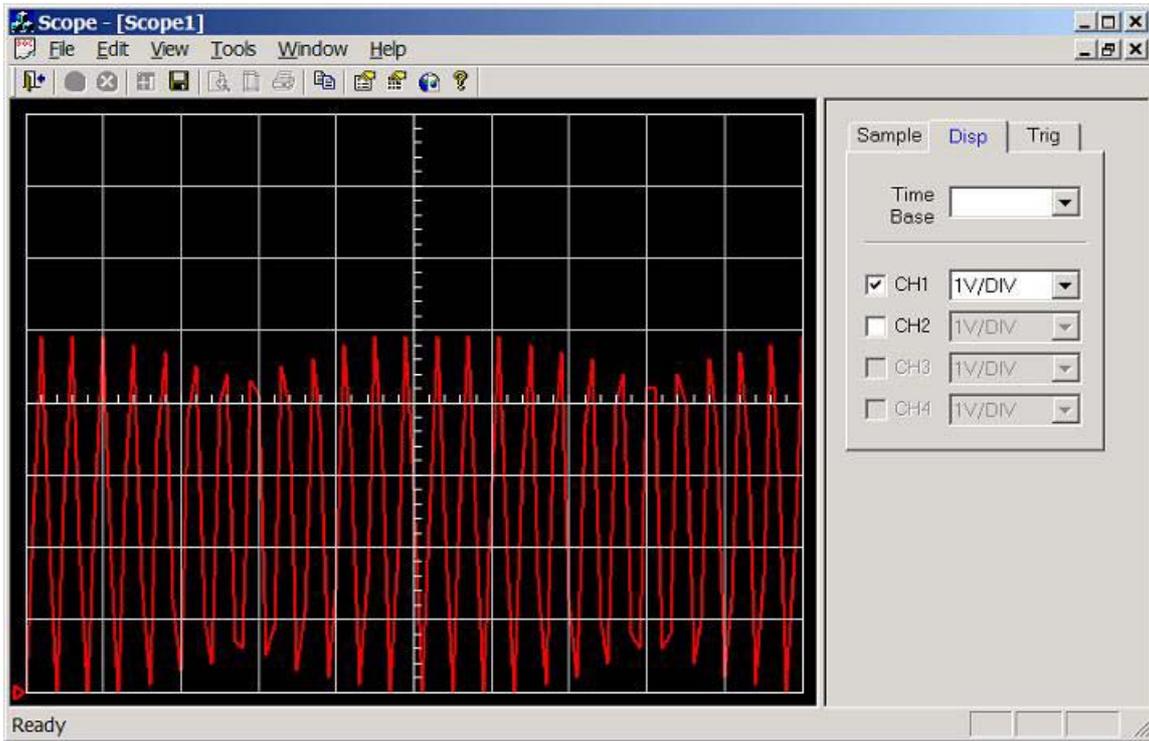


Figure 11.2.3.1g. Screen dump of scope program (sample rate = 2,000Hz, input wave = 500Hz sinewave 0-5V)

1,000Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3.1h (resolution 100 x 80, that's 10 x 8 divisions), something real strange has happened here, clearly the result of under sampling.

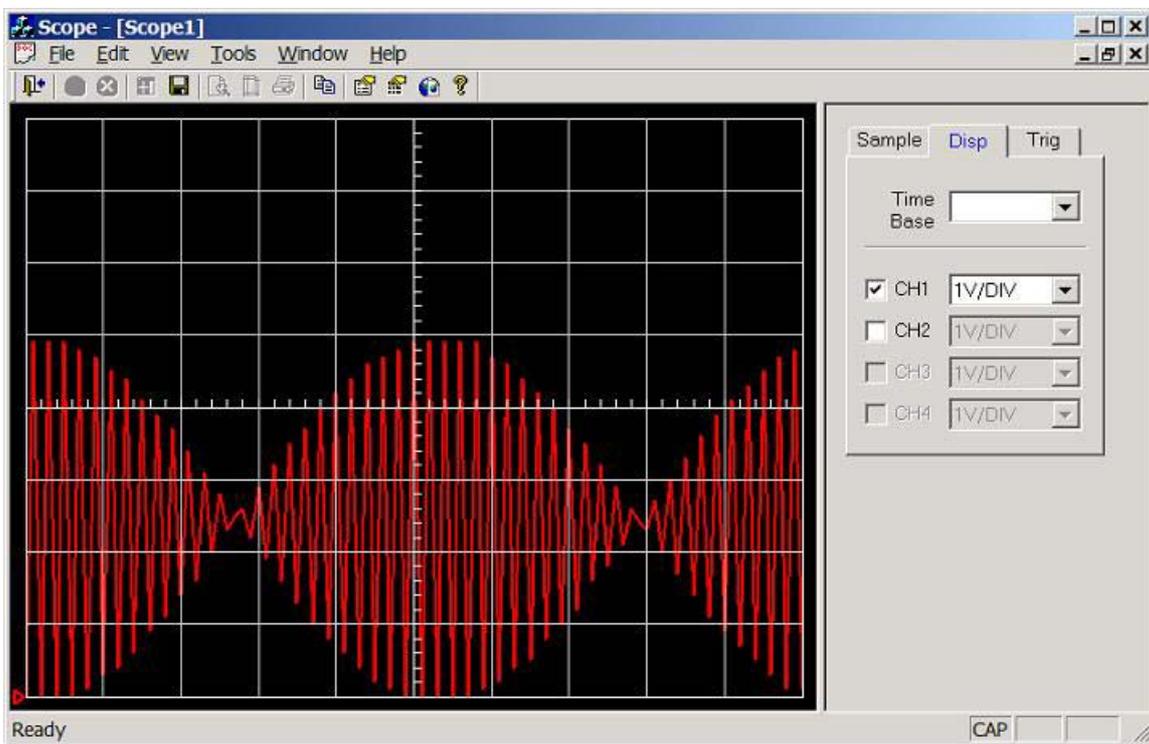


Figure 11.2.3.1h. Screen dump of scope program (sample rate = 2,000Hz, input wave = 1,00Hz sinewave 0-5V)

20Hz trianglewave (0-5V) connected to CH1/AN0, see figure 11.2.3.1i (resolution 200 x 160, that's 20 x 16 divisions), 9.9 division per cycle, hence the calculated frequency of the waveform is $1/(9.9 * 0.005) = 20.2\text{Hz}$.

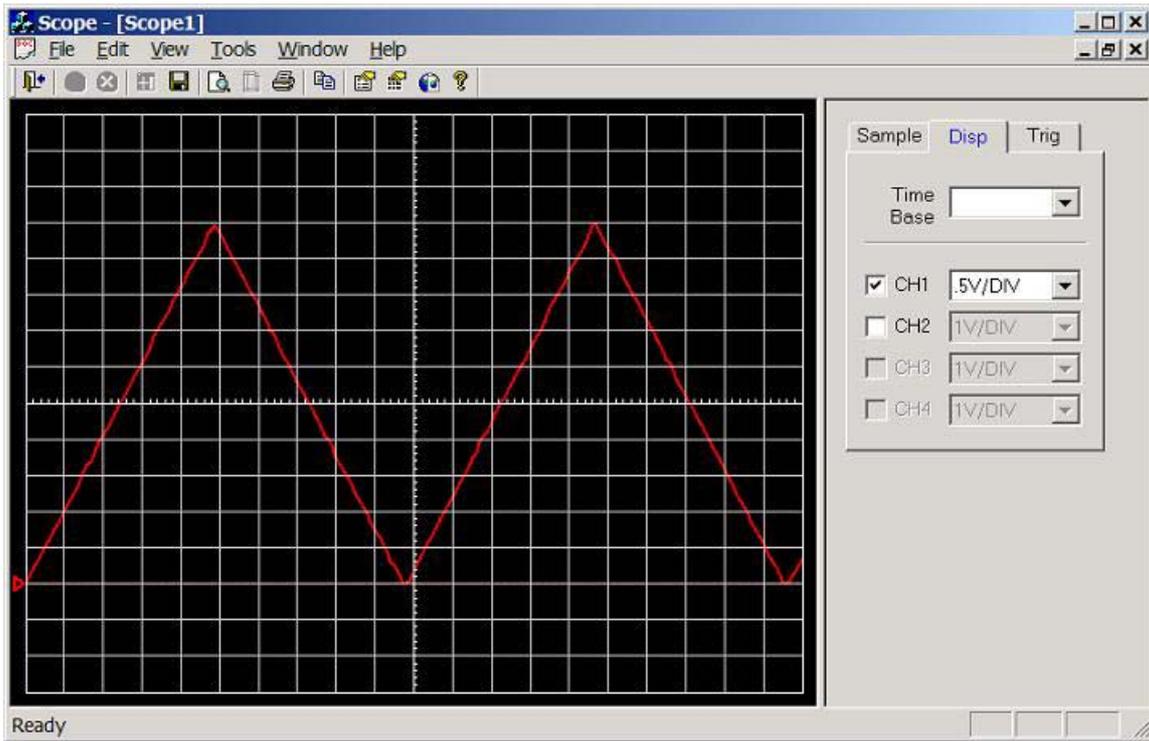


Figure 11.2.3.1i. Screen dump of scope program (sample rate = 2,000Hz, input wave = 20Hz trianglewave 0-5V)

20Hz squarewave (0-5V) connected to CH1/AN0, see figure 11.2.3.1j (resolution 200 x 160, that's 20 x 16 divisions), 9.9 division per cycle, hence the calculated frequency of the waveform is $1/(9.9 * 0.005) = 20.2\text{Hz}$.

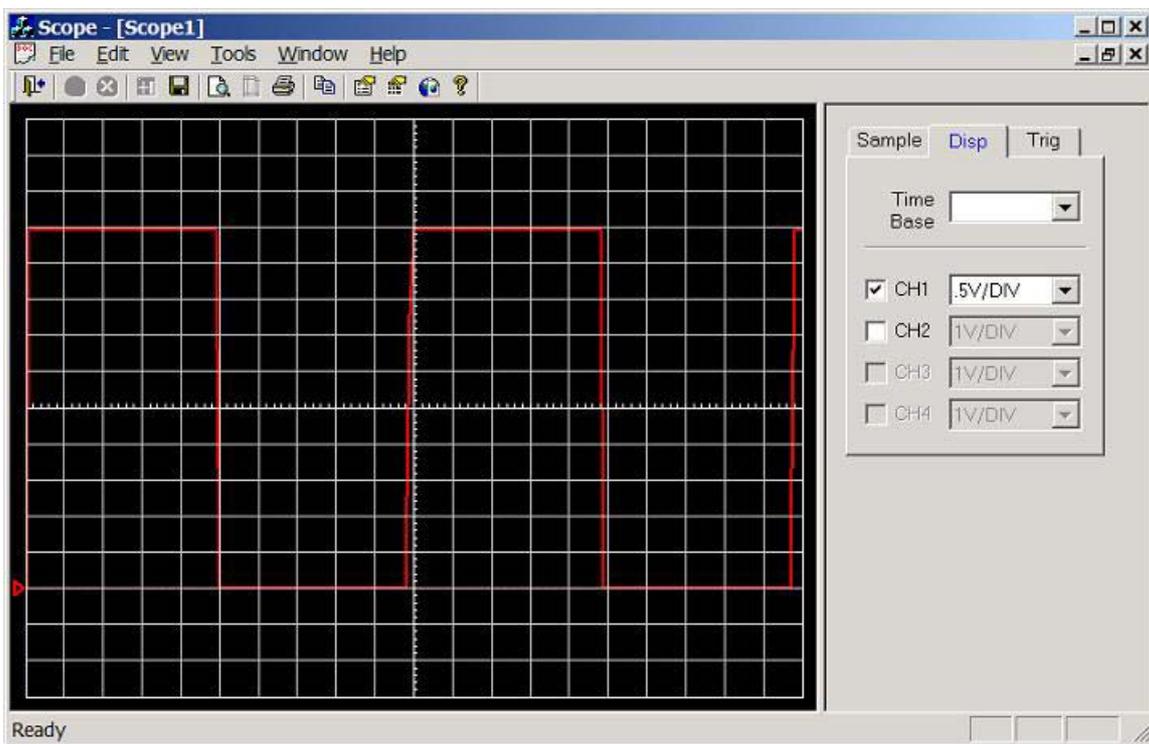


Figure 11.2.3.1j. Screen dump of scope program (sample rate = 2,000Hz, input wave = 20Hz squarewave 0-5V)

11.2.3.2. Sample rate changed to 5000Hz (mark6c.c)

Hence scope time-base is 0.002 seconds per division that's 500Hz.

50Hz sinewave (0-5V) connected to CH1/AN0, see figure 11.2.3.2a (resolution 200 x 160, that's 20 x 16 divisions), 8 division per cycle, hence the calculated frequency of the waveform is $1/(8 * 0.002) = 62.5\text{Hz}$. Clearly something bad has happened here; there are gaps in the data. The reason for this is that the UART transmit buffer fills up and the program must wait until the transmit buffer is free to accept another character; hence there is an additional time delay.

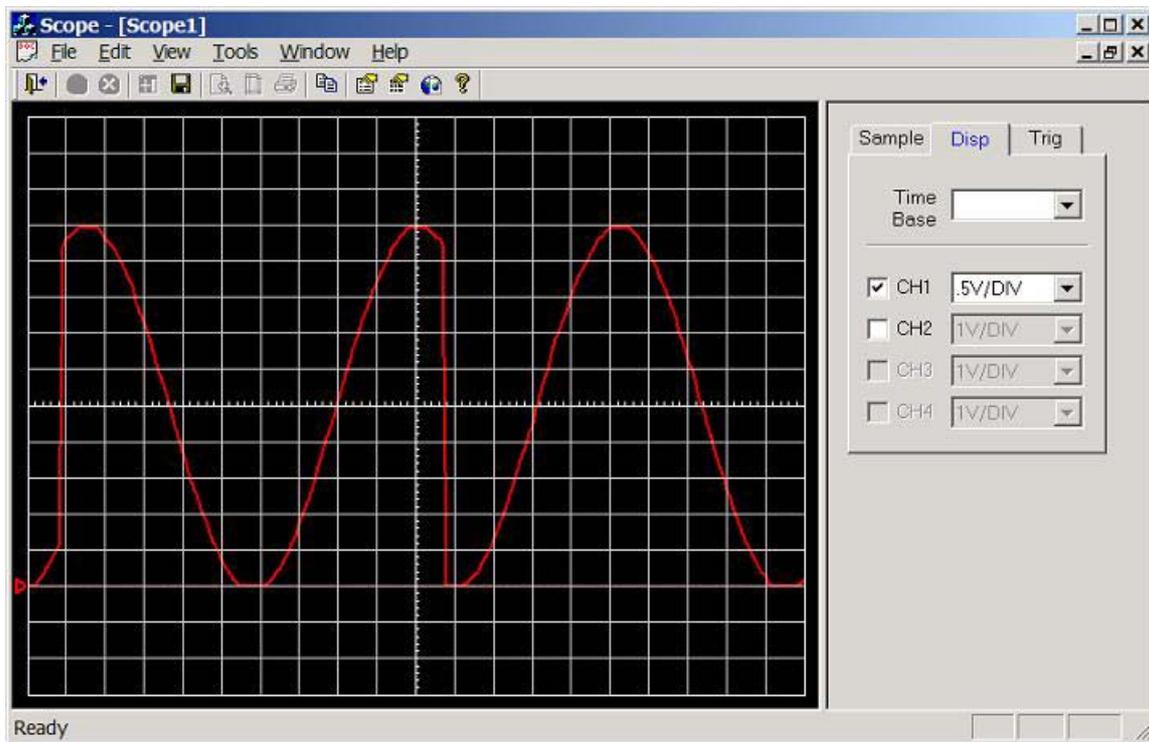


Figure 11.2.3.2a. Screen dump of scope program (sample rate = 5,000Hz, input wave = 50Hz squarewave 0-5V)

This program reads the ADC and transmits two characters with a $200\mu\text{s}$ delay before the process repeats over and over again. But if the transmits buffer is full, note that for 115kbps it takes $8.7\mu\text{s}$ ($1/115,000$) to transmit each character that's $86.9\mu\text{s}$ per byte. Using the PICs built in timers should fix this problem.

But the main reason for the distorted waveform was because the PC serial buffer was overflowing. The incoming characters were coming in faster than the scope program was processing them hence the buffer was overflowing, causing gaps to appear in the data. The solution is to increase the scope programs communication protocol real-time timer (e.g. from 100ms to 25ms), and increase the buffer size (from 1k to 10k).

11.2.4. Test 7: Test Timer Interrupts.

Program mark7.c was compiled and loaded into the PIC using Microchip MPLAB. This program is simple, basically it setups all of the PIC onboard timers to interrupt the main program at preset timer intervals, then "Main..." is transmitted once every second forever. Timer 0 interrupt is used to transmit "Interrupt_T0" twice a second, timer 1 interrupt is used to transmit "Interrupt_T1" 4-times a second and timer 2 interrupt is used to transmit "Interrupt_T2" 10-times a second. The DOS based test program (test.exe, version 1.05) was used in the RX terminal mode to receive incoming characters. This test fully tests the timer interrupts, which will be used for time base, communications etc...

Screen dump of text.exe after a couple of seconds: -

```
Waiting for message (press a key to stop)
Main...Interrupt_T2...Interrupt_T2...Interrupt_T1...Interrupt_T2...Interrupt_T2.
..Interrupt_T2...Interrupt_T0...Interrupt_T1...Interrupt_T2...Interrupt_T2...Int
errupt_T2...Interrupt_T1...Interrupt_T2...Interrupt_T2...Interrupt_T0...Interrup
t_T1...Interrupt_T2...Main...Interrupt_T2...Interrupt_T2...Interrupt_T1...Interr
upt_T2...Interrupt_T2...Interrupt_T2...Interrupt_T0...Interrupt_T1...Interrupt_T
```

```

2...Interrupt_T2...Interrupt_T2...Interrupt_T1...Interrupt_T2...Interrupt_T2...I
nterrupt_T0...Interrupt_T1...Interrupt_T2...Main...Interrupt_T2...Interrupt_T2..
..Interrupt_T1...Interrupt_T2...Interrupt_T2...Interrupt_T2...Interrupt_T0...Inte
rrupt_T1...Interrupt_T2...Interrupt_T2...Interrupt_T1...Interrupt_T2...Interrupt
_T2...Interrupt_T2...Interrupt_T0...Interrupt_T1...Interrupt_T2...Main...Interru
pt_T2...Interrupt_T2...Interrupt_T1...Interrupt_T2...Interrupt_T2...Interrupt_T2
...Interrupt_T0...Interrupt_T1...Interrupt_T2...Interrupt_T2...Interrupt_T1...In
terrupt_T2...Interrupt_T2...Interrupt_T2...Interrupt_T0...Interrupt_T1...Interru
pt_T2...Interrupt_T2...Main...Interrupt_T2...Interrupt_T1...Interrupt_T2...Inter
rupt_T2...Interrupt_T0...Interrupt_T1...Interrupt_T2...Interrupt_T2...Interrupt_
T2...Interrupt_T1...Interrupt_T2...Interrupt_T2...Interrupt_T2...Interrupt_T0...
Interrupt_T1...Interrupt_T2...Interrupt_T2...Main...Interrupt_T1...Interrupt_T2..
..Interrupt_T2...Interrupt_T2...Interrupt_T0...Interrupt_T1...Interrupt_T2...Int
errupt_T2...Interrupt_T2...Interrupt_T1...Interrupt_T2...Interrupt_T2...Interrup
t_T2...Interrupt_T0...Interrupt_T1...Interrupt_T2...Interrupt_T2...Interrupt_T1.
..Main...Interrupt_T2...Interrupt_T2...Interrupt_T2...Interrupt_T0...Interrupt_T
1...Interrupt_T2...Interrupt_T2...Interrupt_T2...Interrupt_T1...Interrupt_T2...I
nterrupt_T2...Interrupt_T0...Interrupt_T1...Interrupt_T2...Interrupt_T2...Interr
upt_T2...

```

Clearly the timer interrupts are working, note that there were 11 “Interrupt_T2” between “Main...”, hence timing is slightly out (should be 10). Timer 1 and Timer 0 timing were correct, allow it is important to realise that the main program uses a loop delay, this delay does not take into account the delay taken to process the interrupt hence this may explain way timer 2 appear to be slightly out.

11.2.5. Conclusion

Clearly the four tests that were carried out were successful. Allow problems occurred when sampling at 5,000Hz (distorted waveform), clearly there are two main reason for this the first is that the UART transmit buffer fills up and the program must wait until the transmit buffer is free to accepted another character; hence there is an additional time delay. The second is that incoming characters were coming in faster than the scope program was processing them hence the buffer was overflowing, causing gaps to appear in the data. The solution of both is simple; to solve the first problem one of the PICs real-time timers can be used for the time-base, hence any delay caused by the UART will not affect the time-base (timer will continue to count). The second problem can easily be solved by making the scope program read the serial port faster, and increasing its buffer.

11.3. Test Session 3: FG Wilson Lab (22/03/2002)

11.3.1. Test 8: Dual Channel Test

Program mark8.c was compiled and loaded into the PIC. Basically this program chops between channel 1 and channel 2 (e.g. sample CH1, trans, sample CH2, trans, delay, repeat). The test was a complete success the scope.exe program was able to display both channels correctly. Allow big triggering problem, e.g. if channel 1 was selected for trigger it was almost impossible to get a stable channel 2 trace, and if channel 2 was selected for trigger it was difficult to get a stable channel 1 trace. This is a software problem and should be fixable.

Screen dump of the scope program (version 1.036, 19/03/2002) during this dual channel test, is shown in figure 11.3.1a.

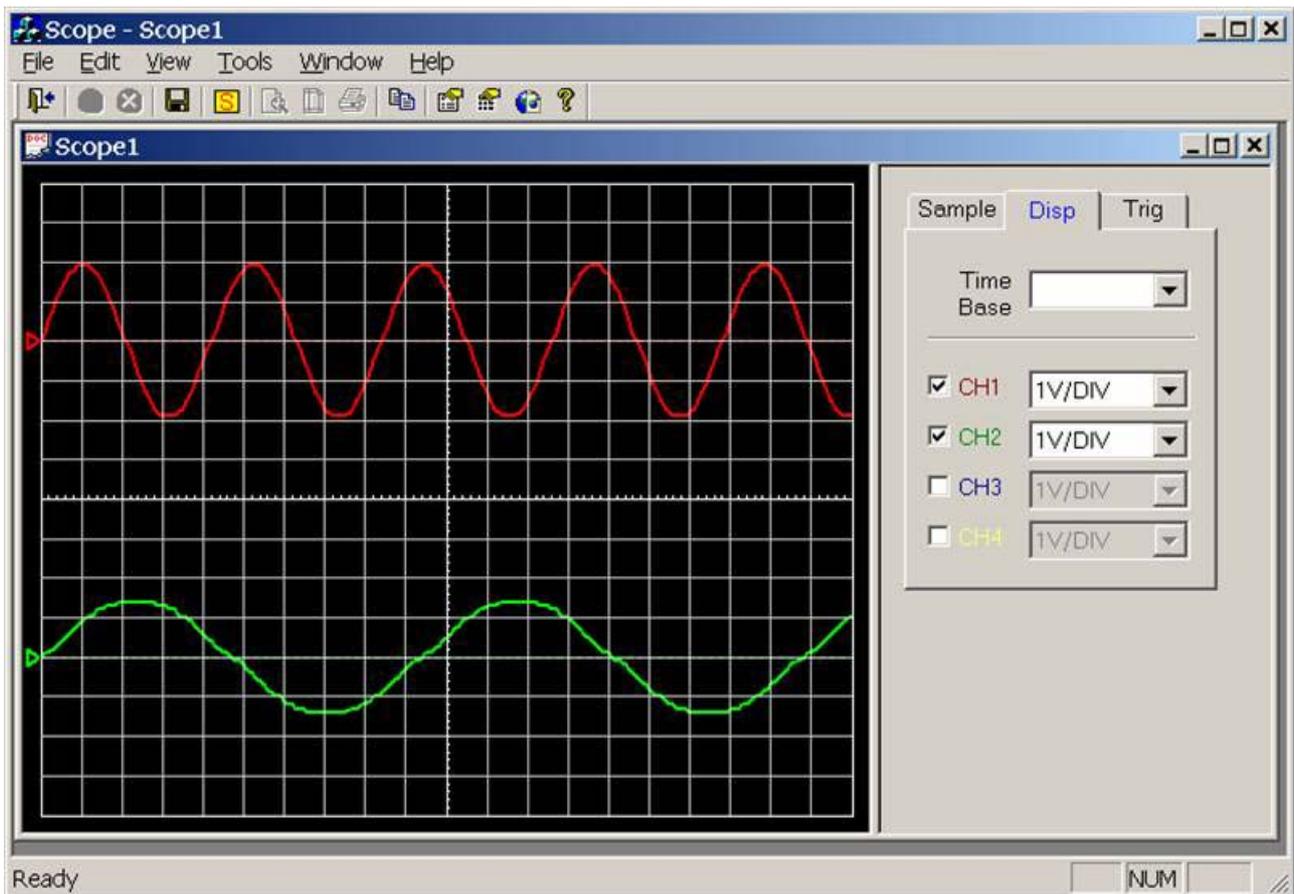


Figure 11.3.1a. Screen dump of scope program (Dual Channel Test)

11.3.2. Test 9: Interrupt Driven Time-Base, Real-Time Sampling (Four Channels)

Program mark9.c was compiled and loaded into the PIC. This program uses timer 2 (PIC built-in timer) as the time-base, the timer causes an interrupt every $20\mu\text{S}$. The variable 'TimeBaseMUX' is the time – base multiplier (e.g. 100 Hz = 10ms, hence TimeBaseMUX = 500). Note no sampling is done during the interrupt routine as all interrupts are disabled while an interrupt is being processed, hence the interrupt routine needs to take less than $20\mu\text{S}$ to execute or timing would be inaccurate (e.g. say the interrupt routine execute time was longer than $20\mu\text{S}$, during this time timer 2 cannot cause any more interrupts, hence accurate timing is lost). The main program loops continuously checking the 'bSample' flag, if the flag is true the function Sample_RealTime() is called, else it continues to loop. The 'bSample' flag is set in the Time-base interrupt routine, keeping the interrupt short.

Interrupt driven time-base has one big advantage over using preset delays: delay routines do not take into consideration processing delays (e.g. waiting for UART buffer to empty), while the PIC's real-time timer will continue to count no matter what the PIC is doing, and will cause an interrupt every $20\mu\text{S}$ (assuming interrupt routine is finished before it is time to call it again).

This program samples all four channels (chop mode). E.g. channel 1 is sampled, then channel 2, then channel 3, then channel 4. It is expected that Alt mode will be added some time in the future, e.g. sample channel 1 a thousand times, then sample channel 2 a thousand times, etc...

The following experiments were carried out with the TimeBaseMux equal to 50, that's a time-base of 1000Hz ($1/(50*20\mu\text{S})$) and baud rate was set at 115kbps. Scope.exe (version 1.036 - 19/03/2002) was used and a calibrated function generator (Fluke PM5139) used to generate input signals.

Note: Sample rate = 1000Hz, 10 horizontal, samples per division. Hence scope time-base is 0.01 seconds per division. Time-base is fixed on this version of the scope program.

25Hz sinewave (0-5V) connected to CH1/AN0 and all other channels connected to 5V, see figure 11.3.2a. 4 divisions per cycle, hence the calculated channel 1 frequency is $1/(4*0.01) = 25\text{Hz}$. Channels 2, 3 & 4 are at 5V DC.

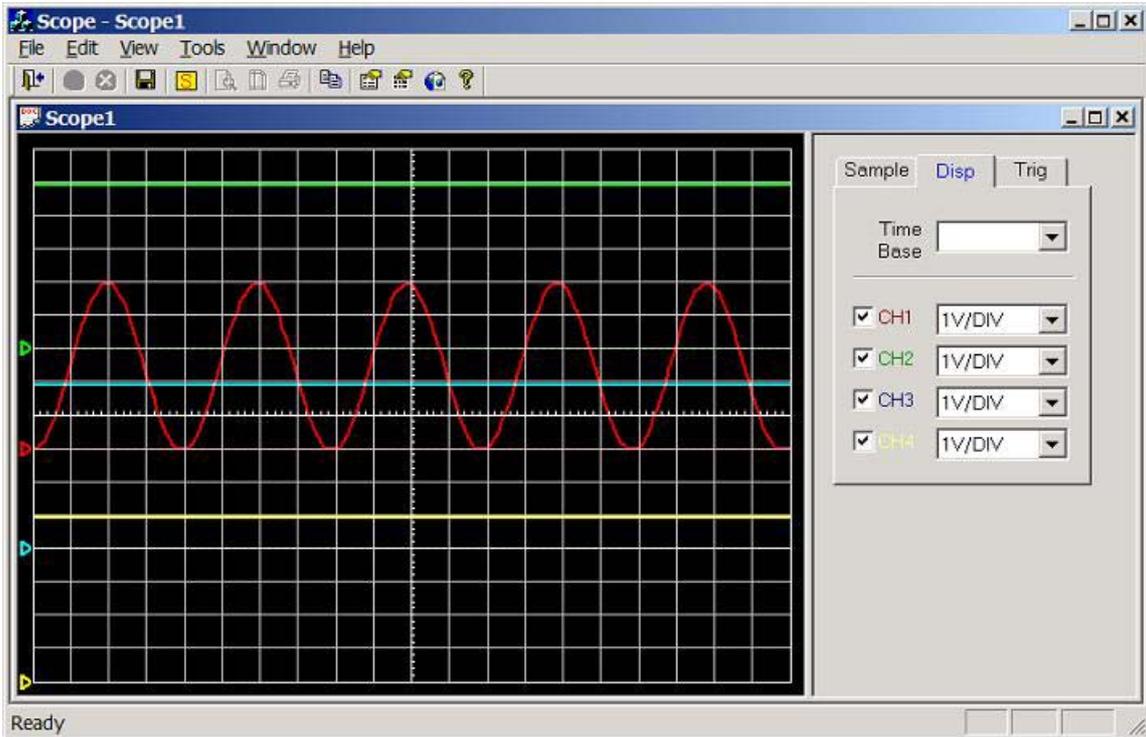


Figure 11.3.2a. Screen dump of scope program (sample rate = 1 kHz, CH1 = 25Hz sinewave, CH2 = 5V, CH3 = 5V, CH4 = 5V)

25Hz sinewave (0-5V) connected to CH2/AN1 and all other channels connected to 5V, see figure 11.3.2b. 4 divisions per cycle, hence the calculated channel 2 frequency is $1/(4*0.01) = 25\text{Hz}$. Channels 1, 3 & 4 are at 5V DC.

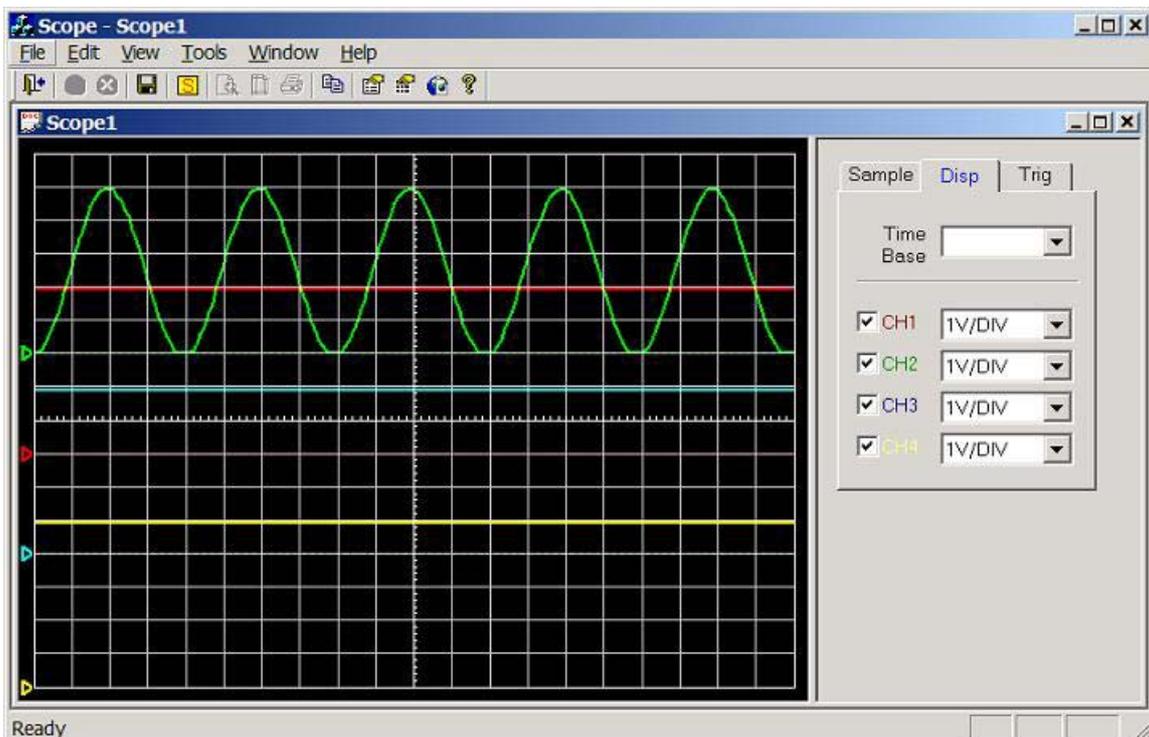


Figure 11.3.2b. Screen dump of scope program (sample rate = 1 kHz, CH2 = 25Hz sinewave, CH1 = 5V, CH3 = 5V, CH4 = 5V)

25Hz sinewave (0-5V) connected to CH3/AN2 and all other channels connected to 5V, see figure 11.3.2c. 4 divisions per cycle, hence the calculated channel 3 frequency is $1/(4 \times 0.01) = 25\text{Hz}$. Channels 1, 2 & 4 are at 5V DC.

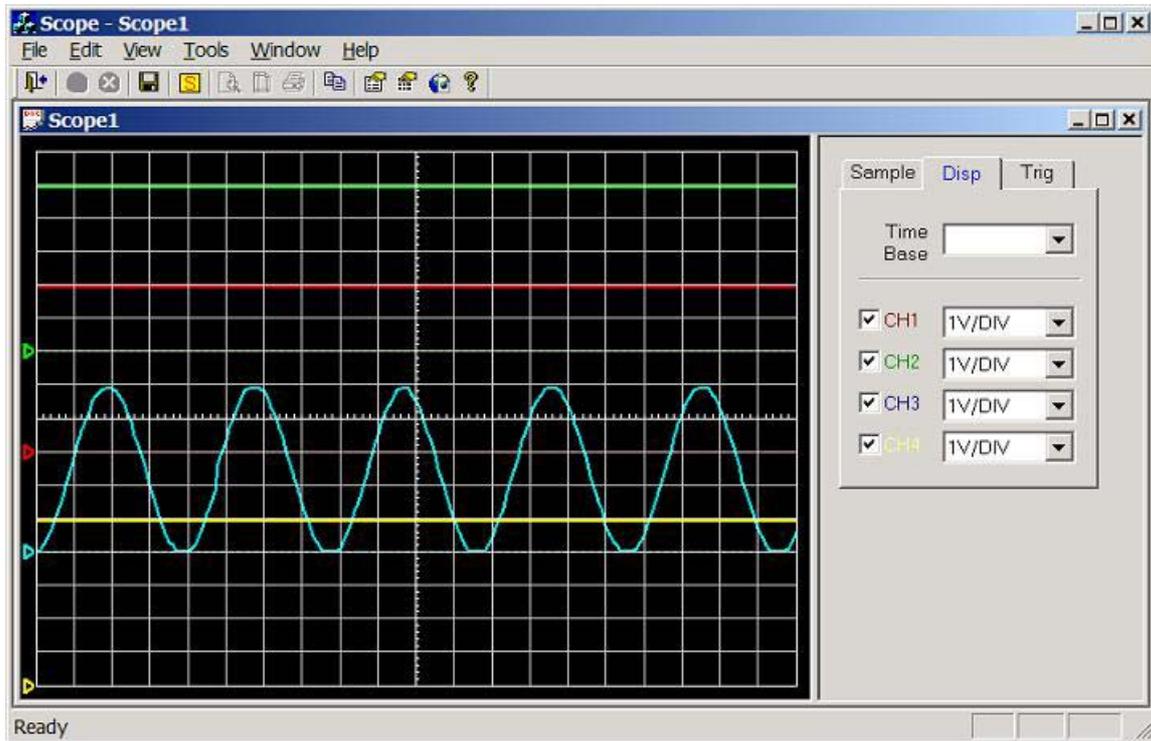


Figure 11.3.2c. Screen dump of scope program (sample rate = 1 kHz, CH3 = 25Hz sinewave, CH1 = 5V, CH2 = 5V, CH4 = 5V)

25Hz sinewave (0-5V) connected to CH4/AN3 and all other channels connected to 5V, see figure 11.3.2d. 4 divisions per cycle, hence the calculated channel 4 frequency is $1/(4 \times 0.01) = 25\text{Hz}$. Channels 1, 2 & 3 are at 5V DC.

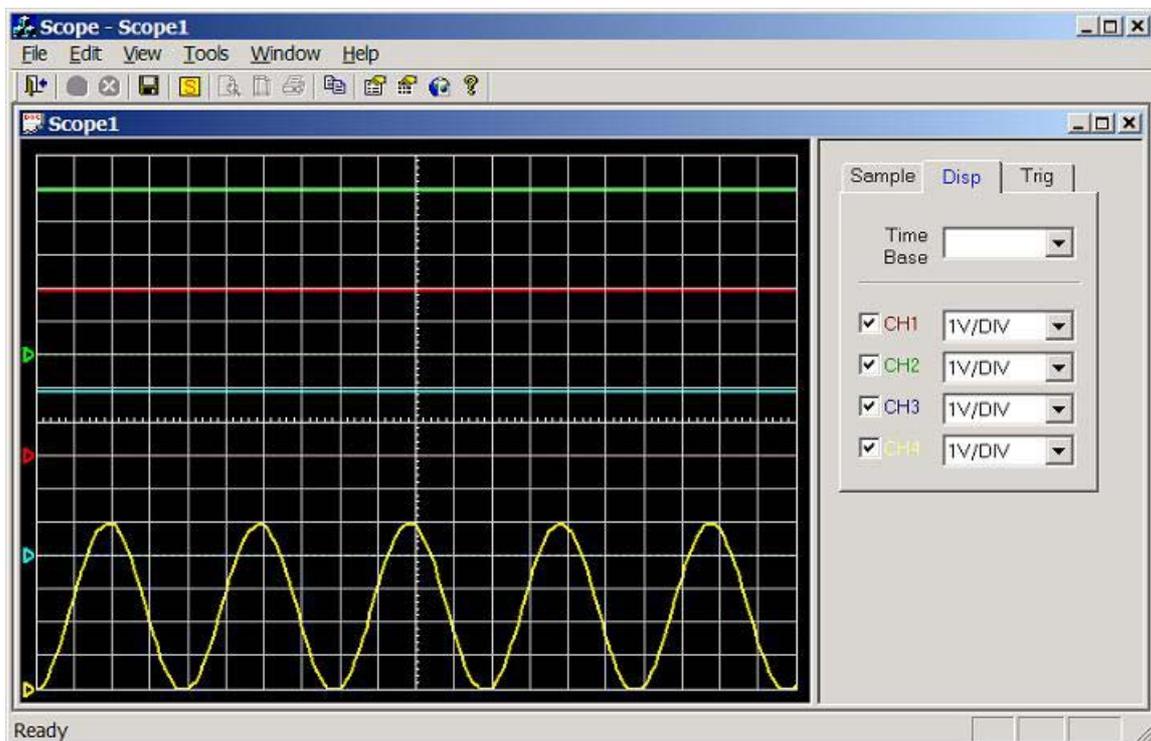


Figure 11.3.2d. Screen dump of scope program (sample rate = 1 kHz, CH4 = 25Hz sinewave, CH1 = 5V, CH2 = 5V, CH3 = 5V)

11.3.2.1. Dual Channel Operation Using Two Signal Generators

Fluke PM5139 function generator (calibrated) connected to **channel 1 (red)**, Thandar TG 101 function generator (not calibrated) connected to **channel 2 (green)**. Sample rate is 1000Hz (scope time-base is 0.01).

Channel 1 connected to 10Hz sine-wave (5Vpp), **channel 2** connected to 30Hz sine-wave (4Vpp), see figure 11.3.2.1a. **Channel 1**: 10 divisions per cycle, hence the calculated frequency is $1/(10 \cdot 0.01) = 10\text{Hz}$. **Channel 2**: 4.2 divisions per cycle, hence the calculated frequency is $1/(4.2 \cdot 0.01) = 23.8\text{ Hz}$ (note channel 2 signal generator is not calibrated, this is the true frequency).

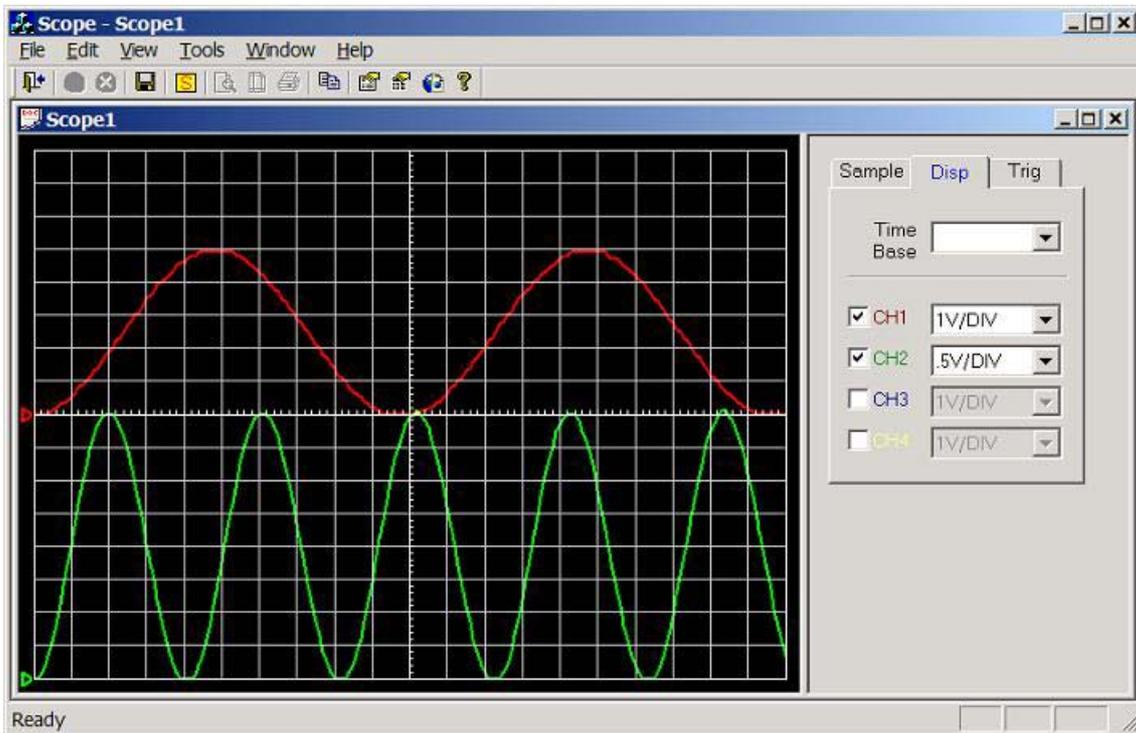


Figure 11.3.2.1a. Screen dump of scope program (sample rate = 1 kHz, CH1 = 10Hz sinewave, CH2 = 30Hz sinewave)

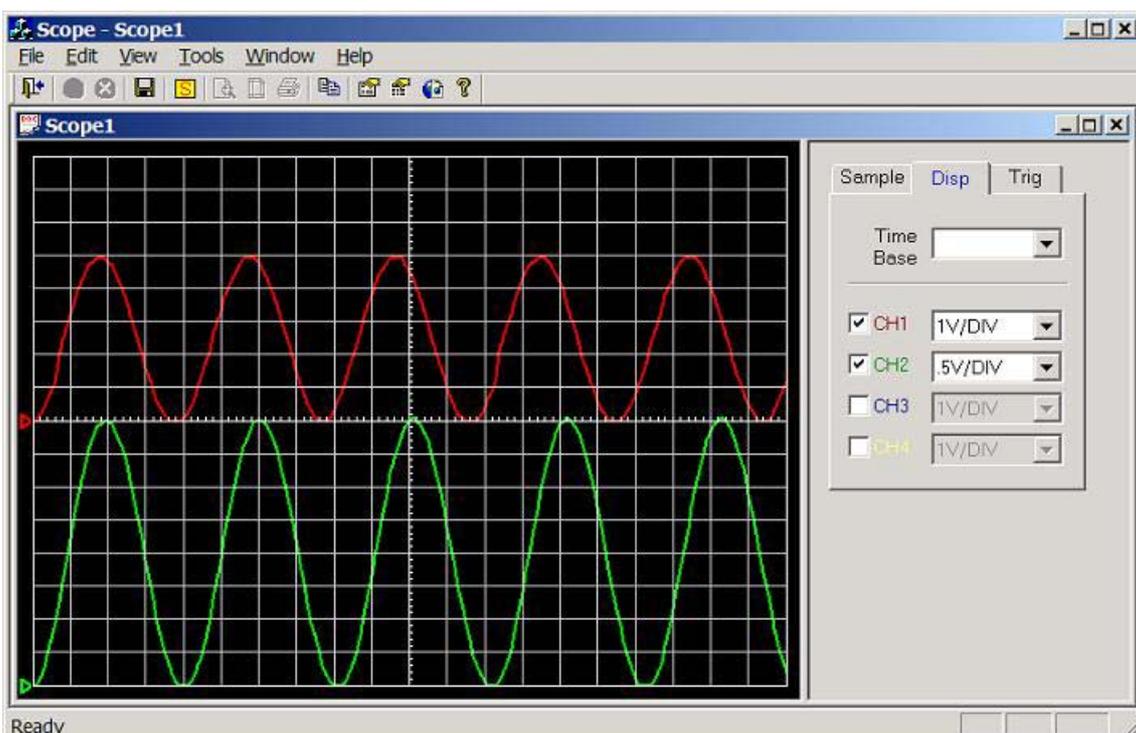


Figure 11.3.2.1b. Screen dump of scope program (sample rate = 1 kHz, CH1 = 25Hz sinewave, CH2 = 30Hz sinewave)

Channel 1 connected to 25Hz sine-wave (5Vpp), channel 2 connected to 30Hz sine-wave (4Vpp), see figure 11.3.2.1b. Channel 1: 4 divisions per cycle, hence the calculated frequency is $1/(4 \times 0.01) = 25\text{Hz}$. Channel 2 unaffected.

Channel 1 connected to 50Hz sine-wave (5Vpp), channel 2 connected to 30Hz sine-wave (4Vpp), see figure 11.3.2.1c. Channel 1: 2 divisions per cycle, hence the calculated frequency is $1/(2 \times 0.01) = 50\text{Hz}$. Channel 2 unaffected.

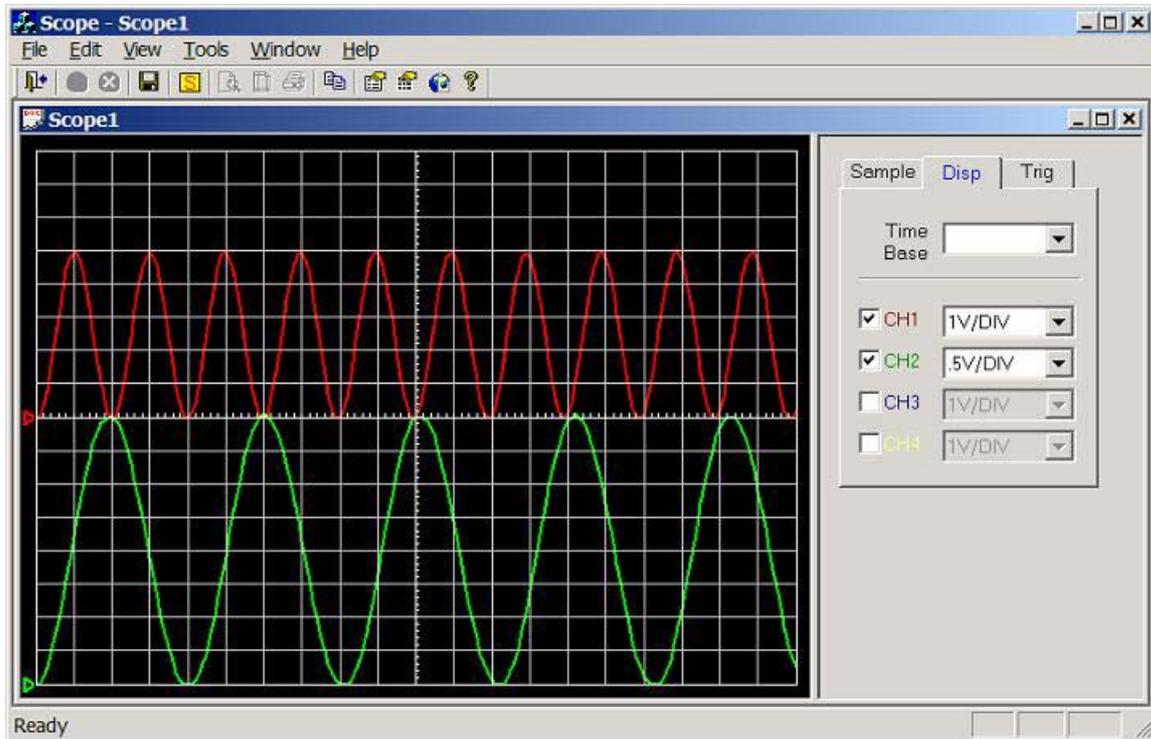


Figure 11.3.2.1c. Screen dump of scope program (sample rate = 1 kHz, CH1 = 50Hz sinewave, CH2 = 30Hz sinewave)

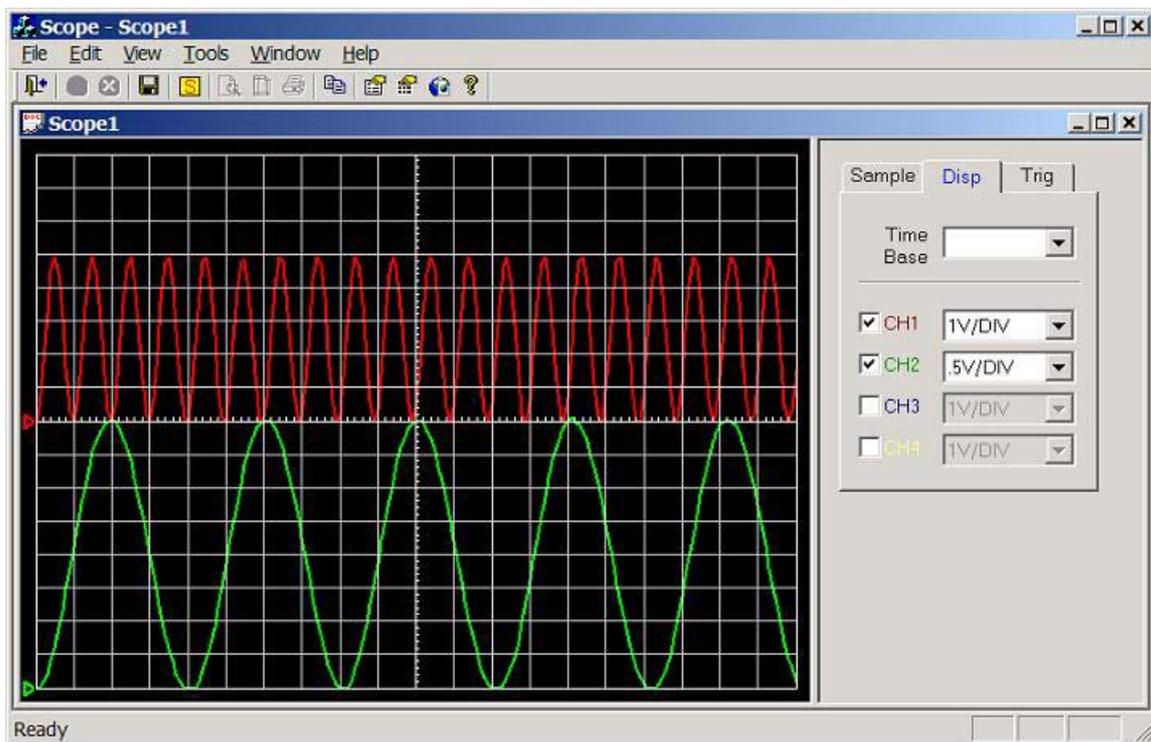


Figure 11.3.2.1d. Screen dump of scope program (sample rate = 1 kHz, CH1 = 100Hz sinewave, CH2 = 30Hz sinewave)

Channel 1 connected to 100Hz sine-wave (5Vpp), channel 2 connected to 30Hz sine-wave (4Vpp), see figure 11.3.2.1d. Channel 1: 1 divisions per cycle, hence the calculated frequency is $1/(1 \cdot 0.01) = 100\text{Hz}$. Channel 2 unaffected.

Channel 1 connected to 500Hz sine-wave (5Vpp), channel 2 connected to 30Hz sine-wave (4Vpp), see figure 11.3.2.1e. Channel 1: Under sampled, aliasing has occurred. Channel 2 unaffected.

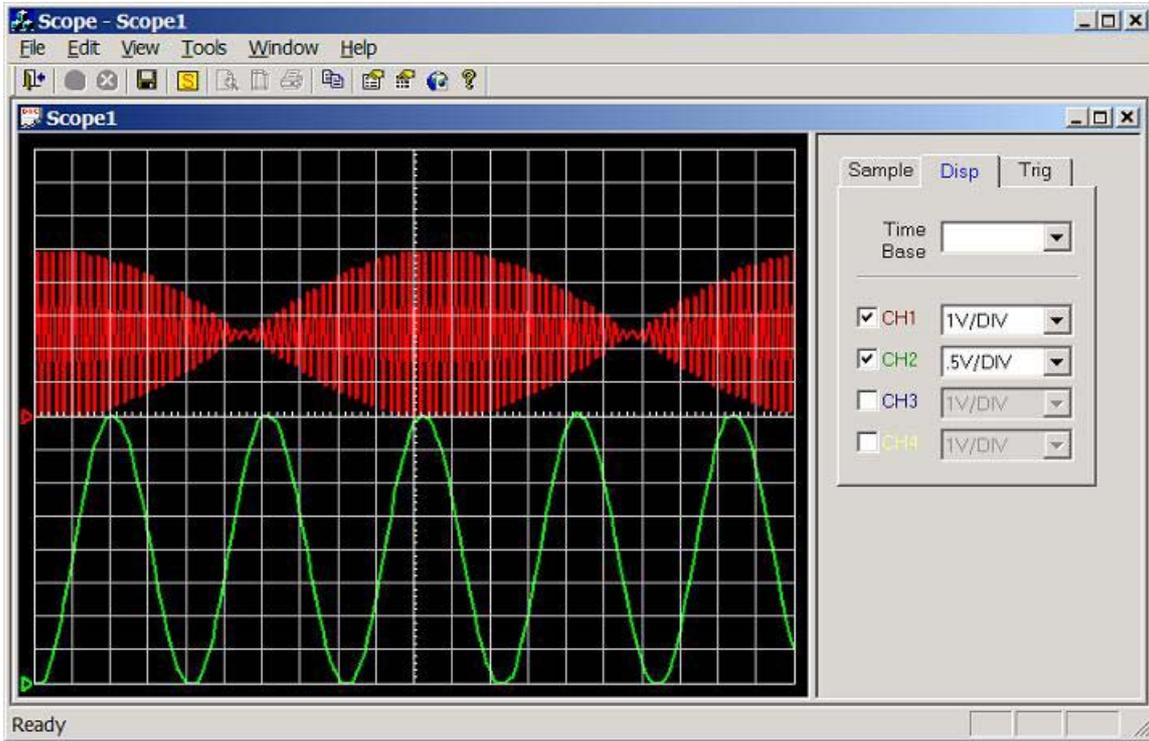


Figure 11.3.2.1e. Screen dump of scope program (sample rate = 1 kHz, CH1 = 500Hz sinewave, CH2 = 30Hz sinewave)

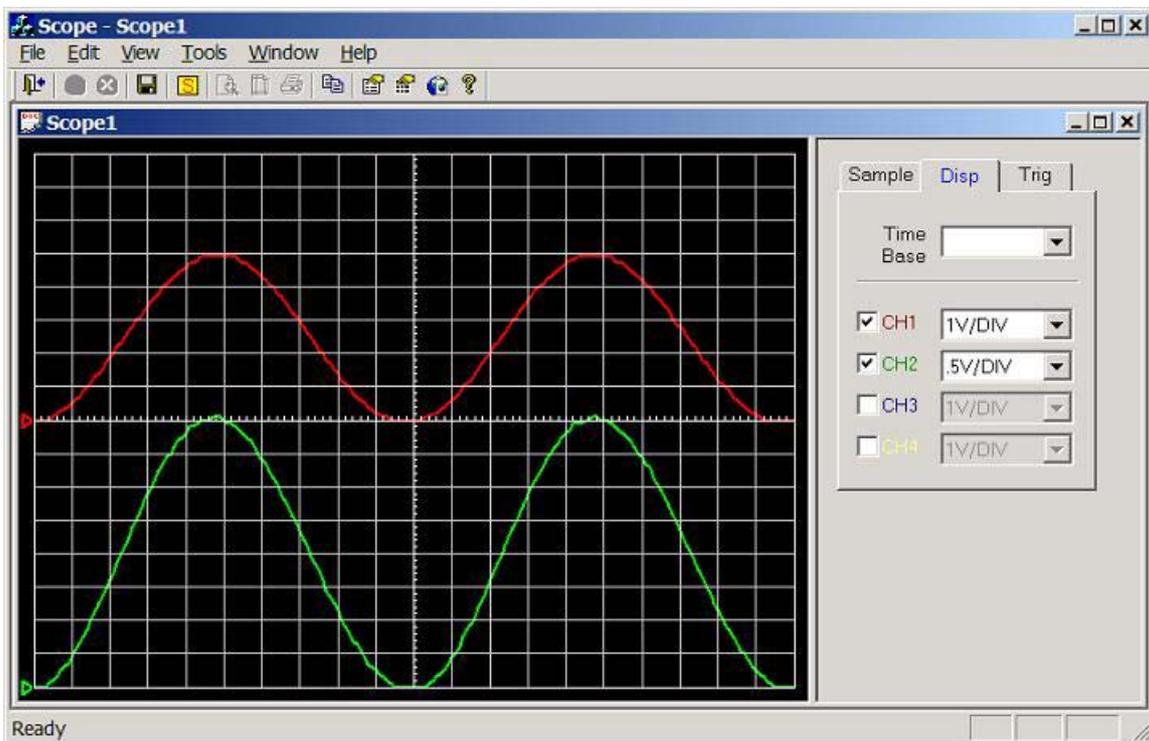


Figure 11.3.2.1f. Screen dump of scope program (sample rate = 1 kHz, CH1 = 500Hz sinewave, CH2 = 30Hz sinewave)

Channel 1 connected to 10Hz sine-wave (5Vpp), channel 2 connected to 10Hz sine-wave (4Vpp), see figure 11.3.2.1f. Channel 1: 10 divisions per cycle, hence the calculated frequency is $1/(10 \cdot 0.01) = 10\text{Hz}$. Channel 2: 4.2 divisions per cycle, hence the calculated frequency is $1/(4.2 \cdot 0.01) = 10\text{ Hz}$

Channel 1 connected to 10Hz sine-wave (5Vpp), channel 2 connected to 50Hz sine-wave (4Vpp), see figure 11.3.2.1g. Channel 1: unaffected. Channel 2: 2 divisions per cycle, hence the calculated frequency is $1/(2 \cdot 0.01) = 50\text{ Hz}$.

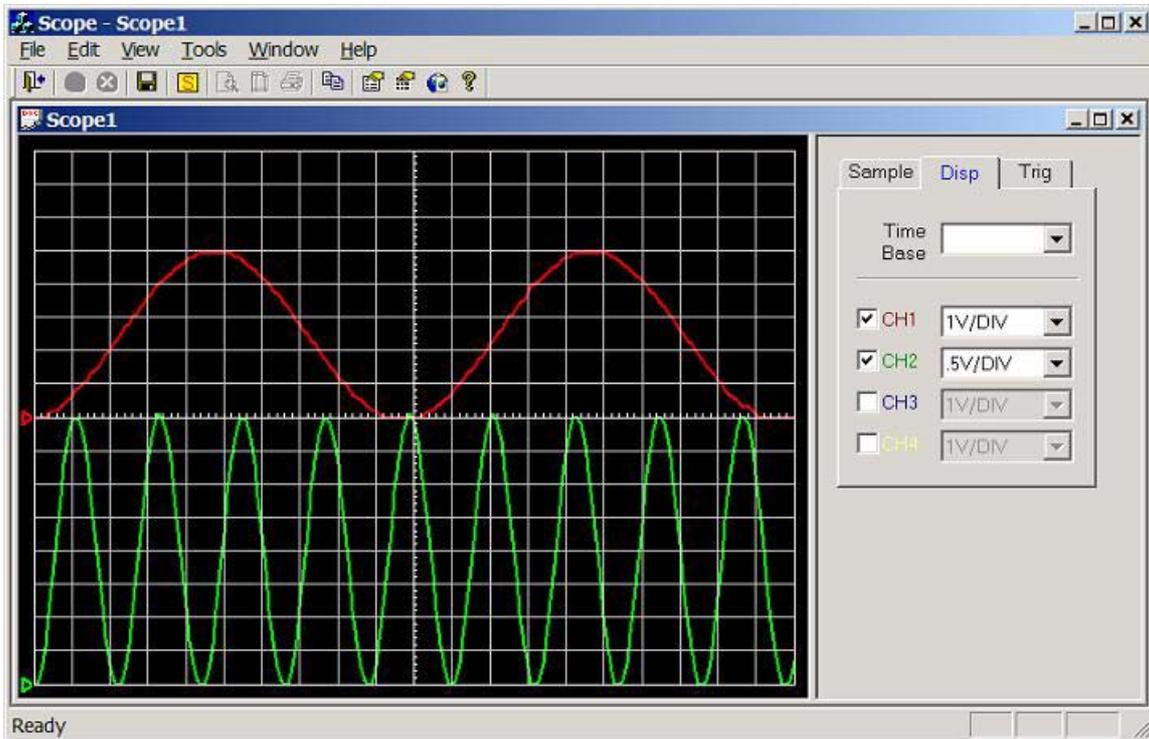


Figure 11.3.2.1g. Screen dump of scope program (sample rate = 1 kHz, CH1 = 10Hz sinewave, CH2 = 50Hz sinewave)

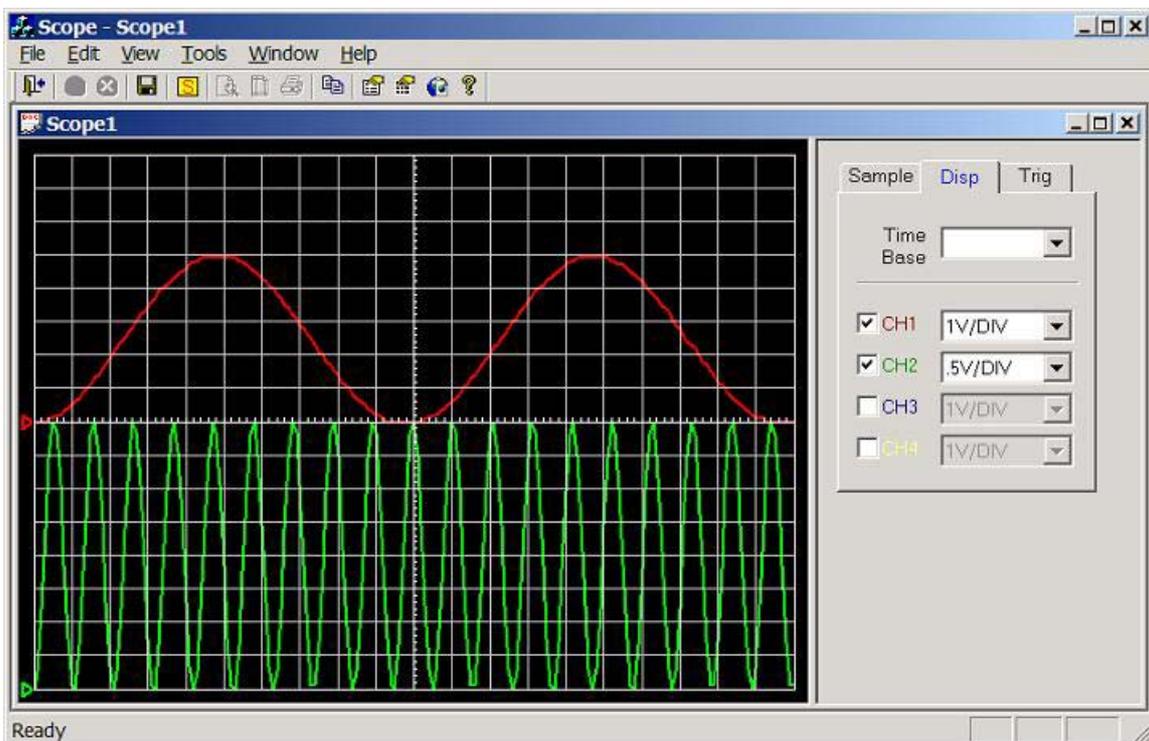


Figure 11.3.2.1h. Screen dump of scope program (sample rate = 1 kHz, CH1 = 10Hz sinewave, CH2 = 100Hz sinewave)

Channel 1 connected to 10Hz sine-wave (5Vpp), channel 2 connected to 100Hz sine-wave (4Vpp), see figure 11.3.2.1h. Channel 1: unaffected. Channel 2: 2 divisions per cycle, hence the calculated frequency is $1/(1*0.01) = 100$ Hz.

Channel 1 connected to 10Hz sine-wave (5Vpp), channel 2 connected to 550Hz sine-wave (4Vpp), see figure 11.3.2.1i. Channel 1: unaffected. Channel 2: Under sampled, aliasing has occurred.

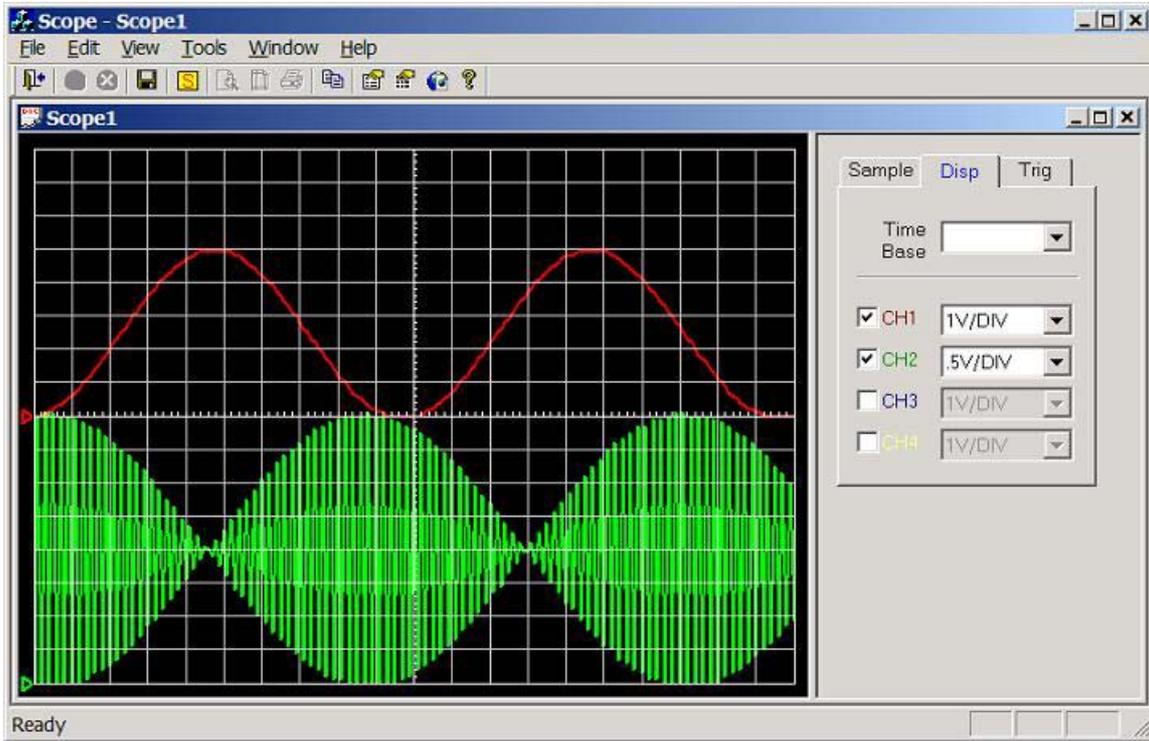


Figure 11.3.2.1i. Screen dump of scope program (sample rate = 1 kHz, CH1 = 10Hz sinewave, CH2 = 550Hz sinewave)

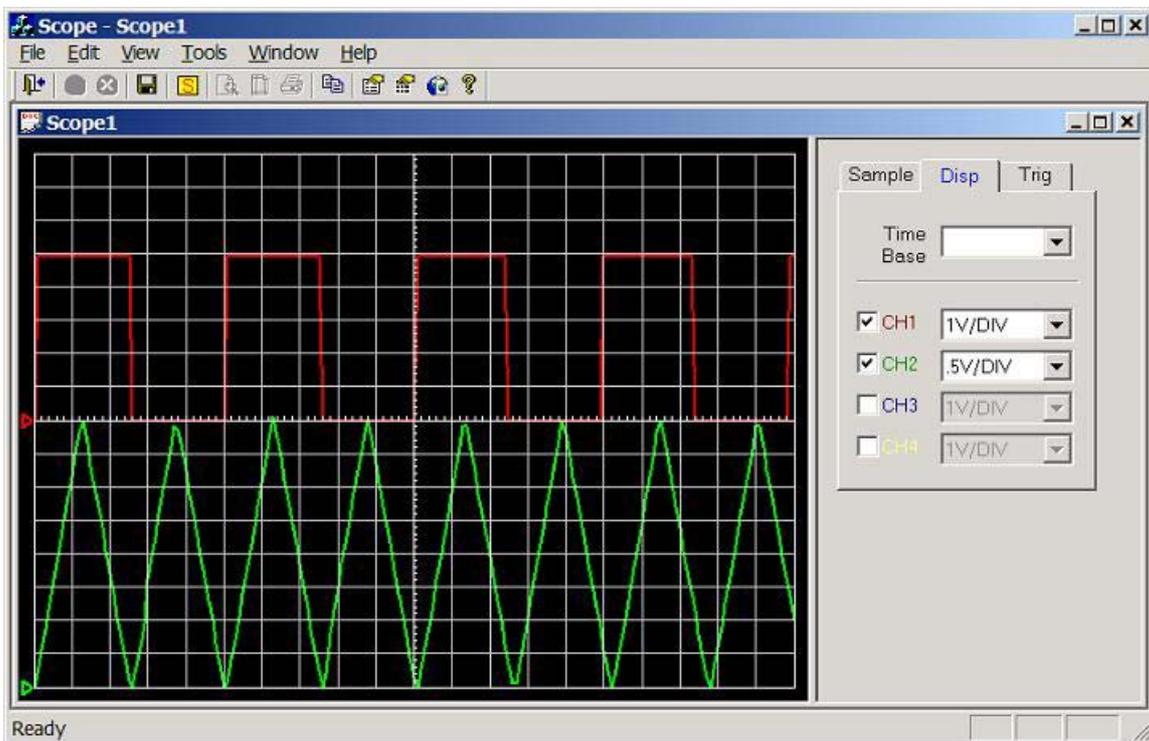


Figure 11.3.2.1j. Screen dump of scope program (sample rate = 1 kHz, CH1 = square wave, CH2 = triangle wave)

Note all of the experiments (in this section) were carried using independent triggering mode, because it was difficult to get two stable traces when triggering off one channel (e.g. if triggering on CH1, CH2 is unstable, and if triggering on CH2, CH1 was unstable). This problem should be software fixable.

11.3.2.2. Quad Channel Operation

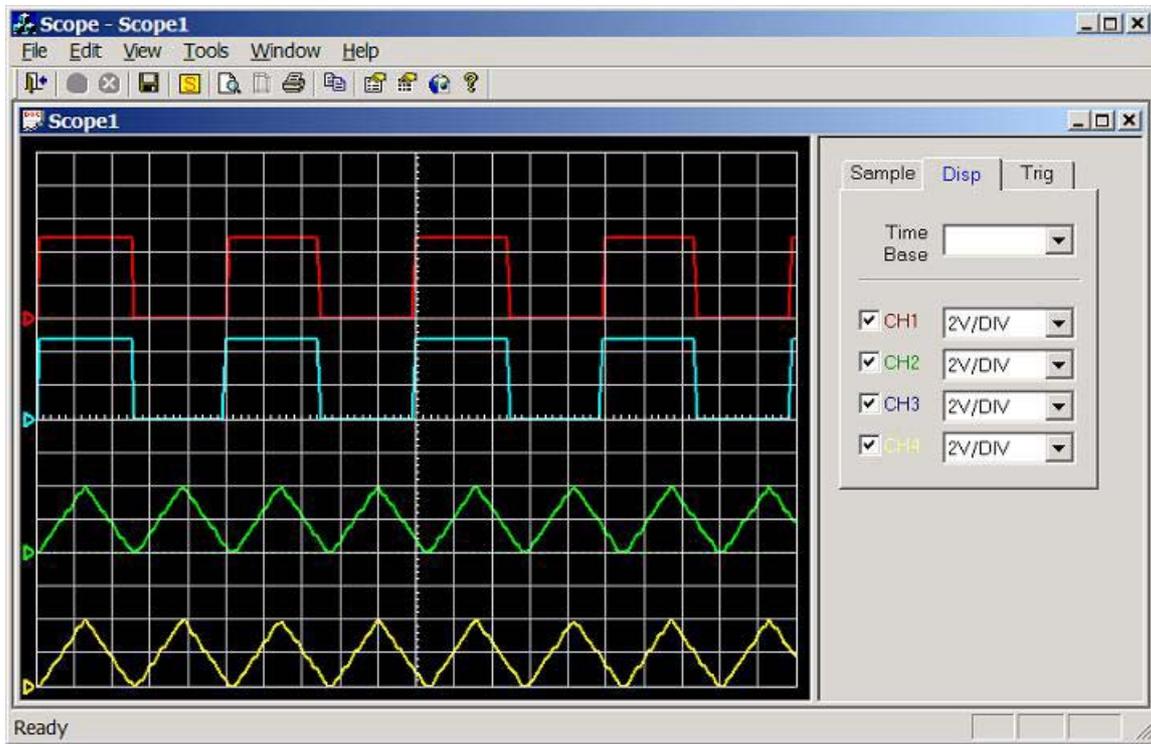


Figure 11.3.2.2a. Screen dump of scope program (CH1 = CH2 = squarewave, CH3 = CH4 = trianglewave)

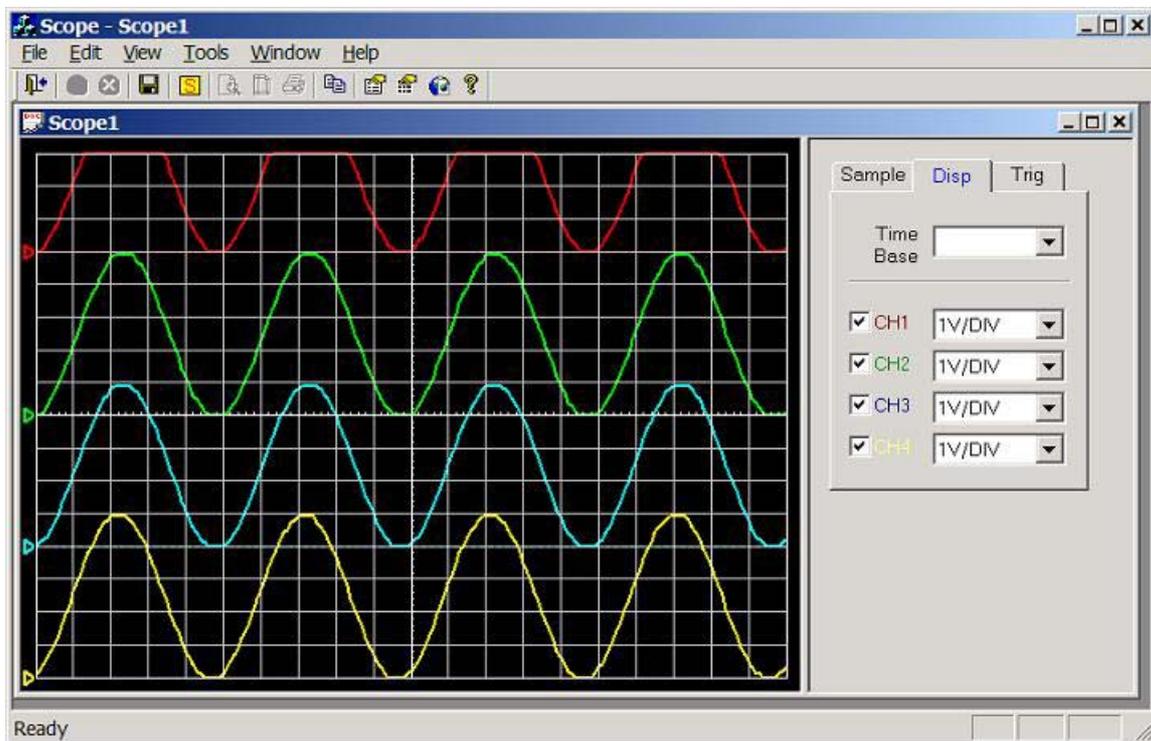


Figure 11.3.2.2b. Screen dump of scope program (CH1 = CH2 = CH3 = CH4 = sinewave, triggering on CH1)

All waveforms in figure 11.3.2.2b were stable, note that there should be a $20\mu\text{S}$ delay (see mark9.c) between each waveform (e.g. chop mode). For example **channel 1** should lead channel 2 by $20\mu\text{S}$, **channel 2** should lead **channel 3** by $20\mu\text{S}$ etc... Each square represents 0.01 seconds (10mS); hence this small offset is not noticeable at this resolution.

11.3.2.3. 5000Hz Sample Rate – Single Channel

mark9.c modified, basically the sample rate has been changed to 5000Hz (TimeBaseMUX = 10), and sampling of channels 2 to 4 has been removed.

Note: Sample rate = 5000Hz, 10 horizontal, samples per division. Hence scope time-base is 2 milliseconds per division. Time-base is fixed on this version of the scope program.

50Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.3.2.3a. 10 divisions per cycle, hence the calculated frequency is $1/(10*0.002) = 50\text{Hz}$.

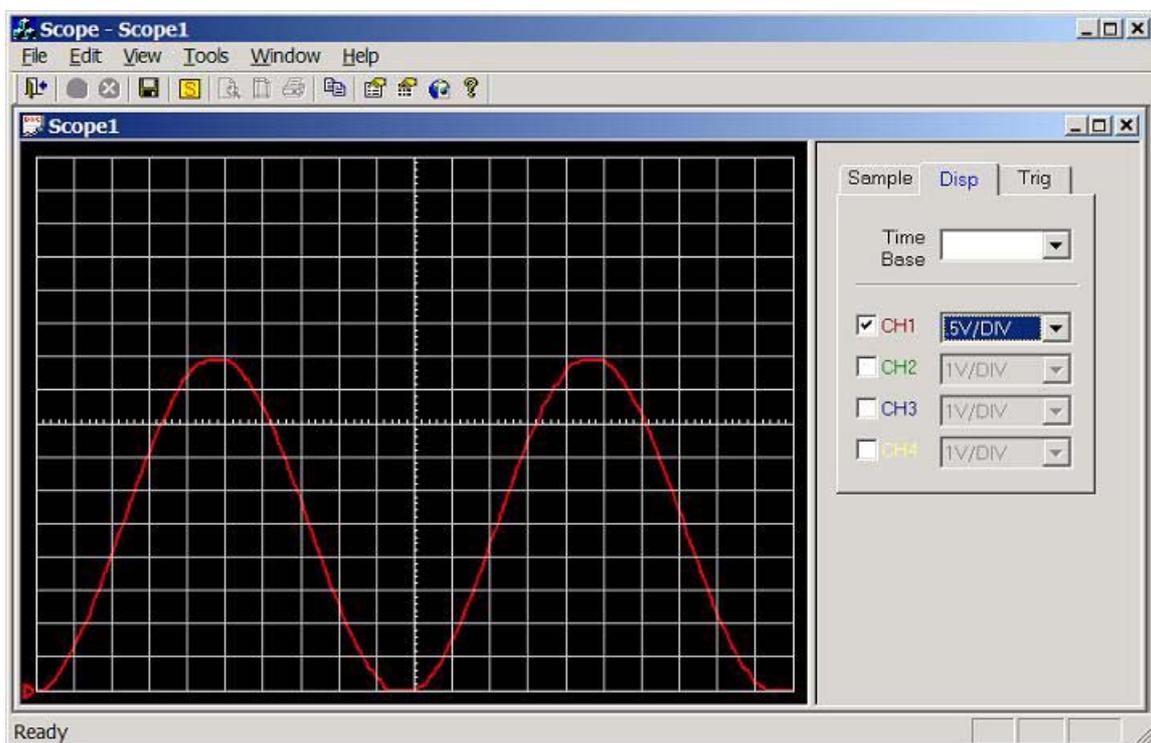


Figure 11.3.2.3a. Screen dump of scope program (sample rate = 5000Hz, input wave = 50Hz sinewave 0-5V)

100Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.3.2.3b. 5 divisions per cycle, hence the calculated frequency is $1/(5*0.002) = 100\text{Hz}$.

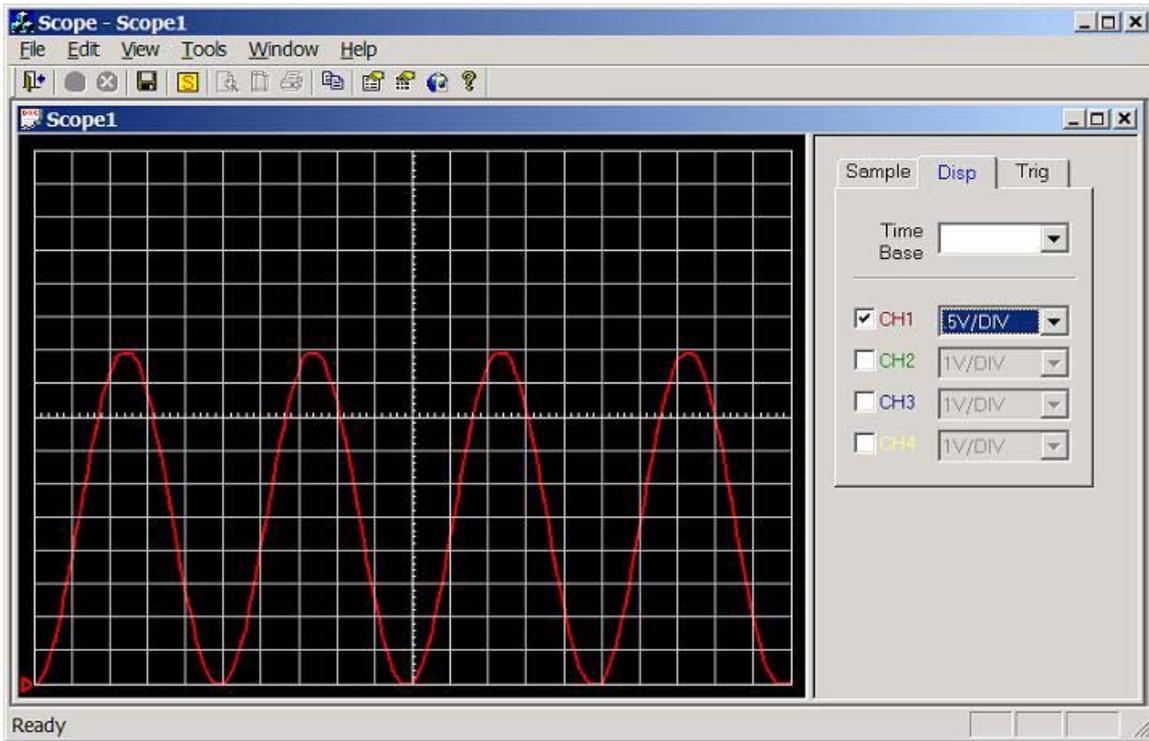


Figure 11.3.2.3b. Screen dump of scope program (sample rate = 5000Hz, input wave = 100Hz sinewave 0-5V)

200Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.3.2.3c. 2.5 divisions per cycle, hence the calculated frequency is $1/(2.5 \cdot 0.002) = 200\text{Hz}$.

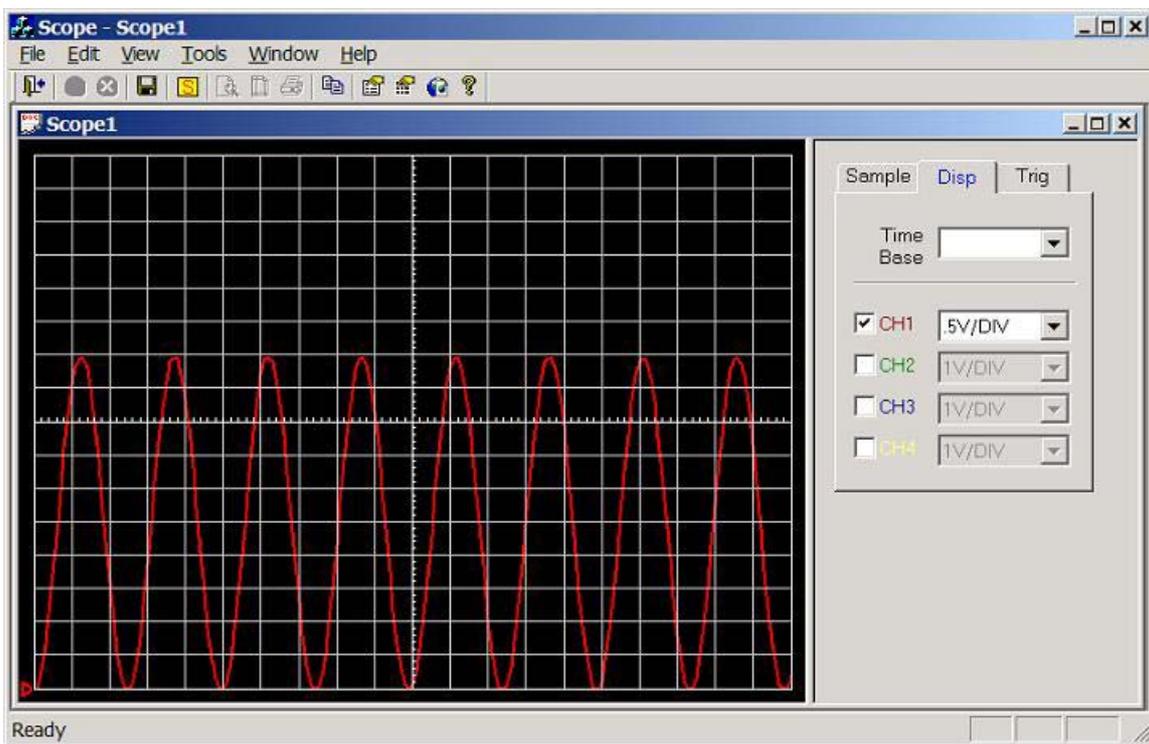


Figure 11.3.2.3c. Screen dump of scope program (sample rate = 5000Hz, input wave = 200Hz sinewave 0-5V)

300Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.3.2.3d. 1.7 divisions per cycle, hence the calculated frequency is $1/(1.7 \cdot 0.002) = 294.1\text{Hz}$.

Figure 11.3.2.3d. Screen dump of scope program (sample rate = 5000Hz, input wave = 300Hz sinewave 0-5V)

400Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.3.2.3e. 1.3 divisions per cycle, hence the calculated frequency is $1/(1.3 * 0.002) = 384.6\text{Hz}$.

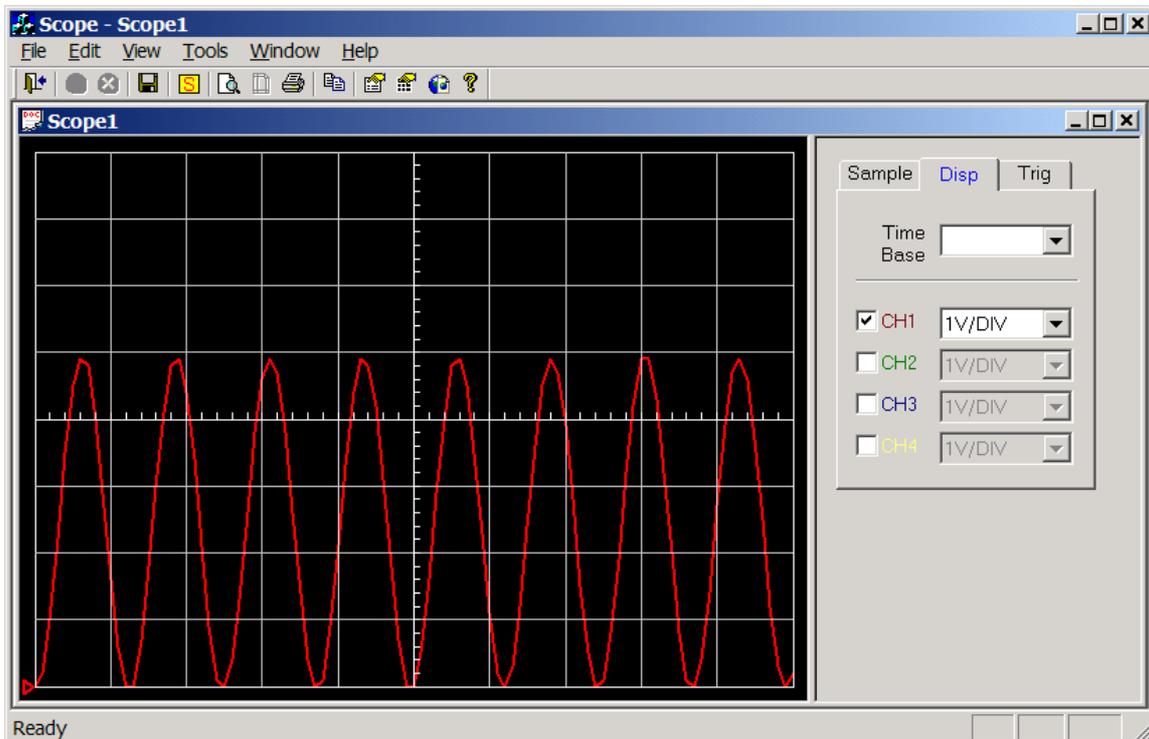


Figure 11.3.2.3e. Screen dump of scope program (sample rate = 5000Hz, input wave = 400Hz sinewave 0-5V)

500Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.3.2.3f. 1 divisions per cycle, hence the calculated frequency is $1/(1 * 0.002) = 500\text{Hz}$.

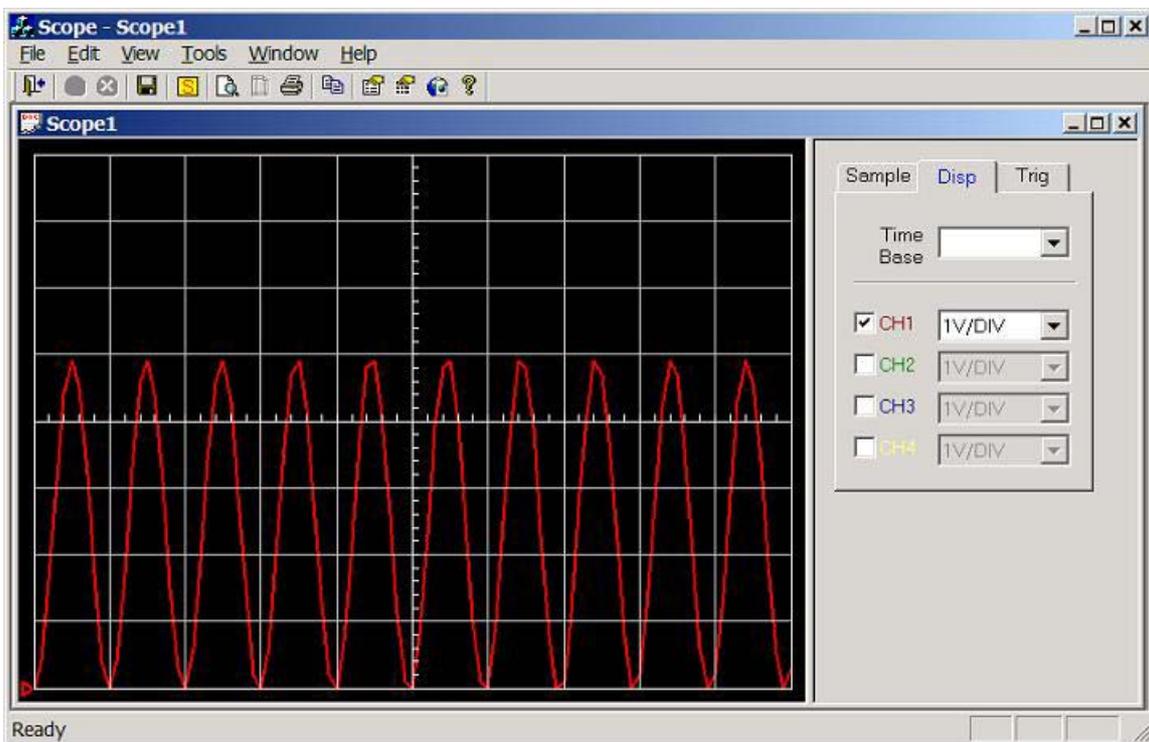


Figure 11.3.2.3f. Screen dump of scope program (sample rate = 5000Hz, input wave = 500Hz sinewave 0-5V)

750Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.3.2.3g. 0.7 divisions per cycle, hence the calculated frequency is $1/(0.7 * 0.002) = 714.2\text{Hz}$.

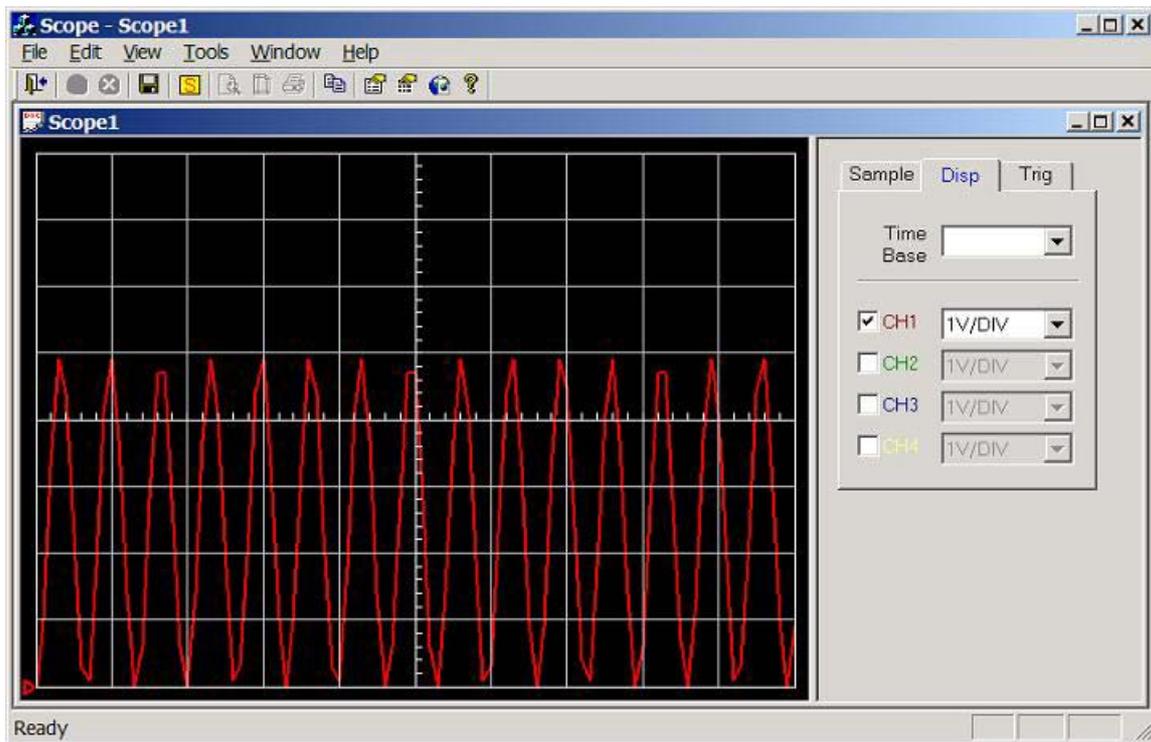


Figure 11.3.2.3g. Screen dump of scope program (sample rate = 5000Hz, input wave = 750Hz sinewave 0-5V)

1000Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.3.2.3h. 0.5 divisions per cycle, hence the calculated frequency is $1/(0.5 * 0.002) = 1000\text{Hz}$, but distorted due to under sampling.

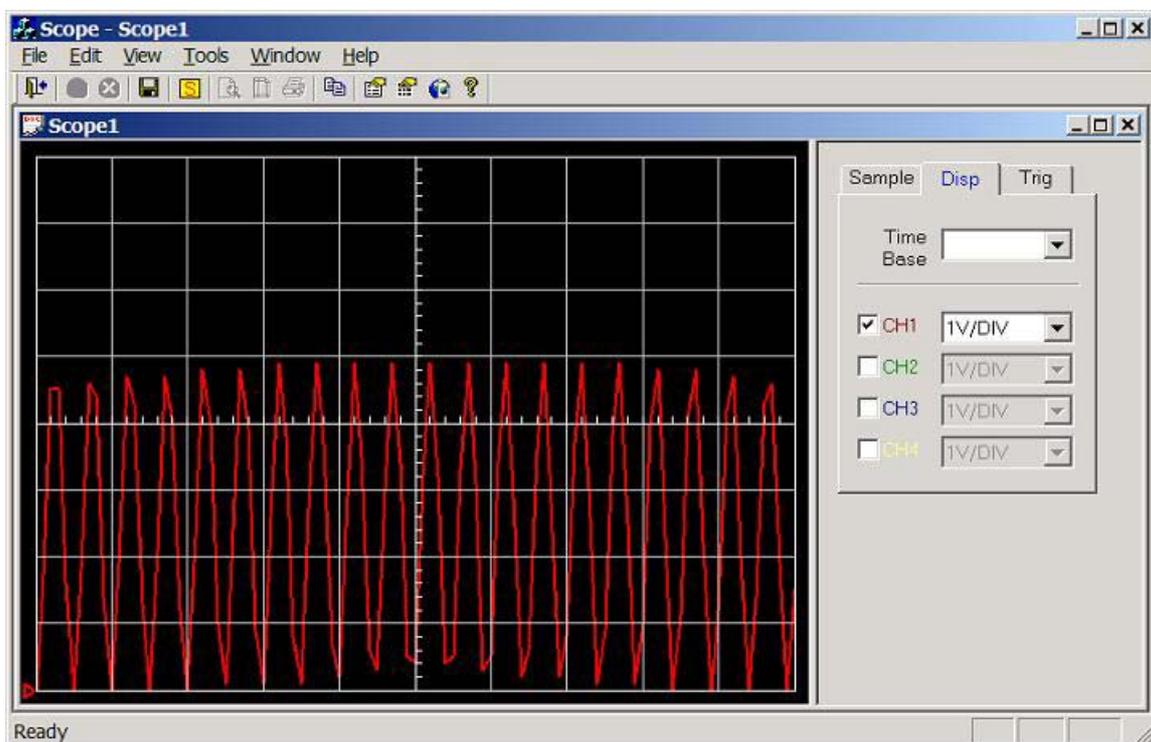


Figure 11.3.2.3h. Screen dump of scope program (sample rate = 5000Hz, input wave = 1,000Hz sinewave 0-5V)

2500Hz sinewave (0-5V) connected to CH1 (AN0), see figure 11.3.2.3i. Under sampled.

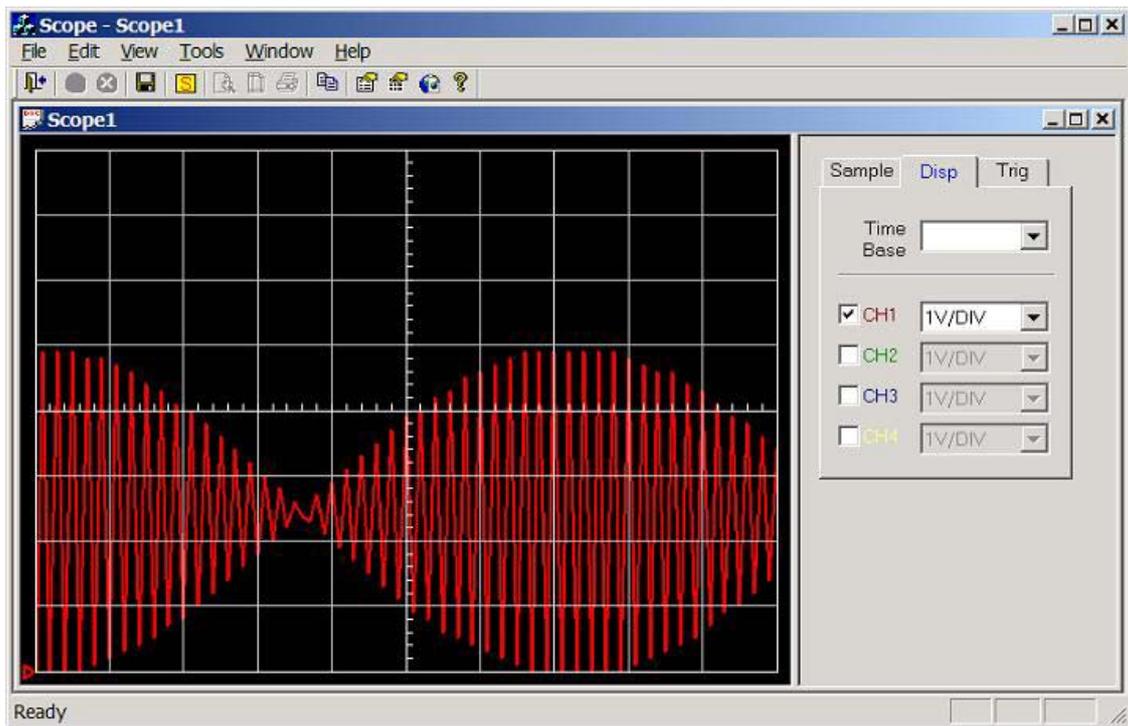


Figure 11.3.2.3i. Screen dump of scope program (sample rate = 5000Hz, input wave = 2,500Hz sinewave 0-5V)

11.3.3. Test 10: Test External RAM chip

The circuit diagram shown in figure 11.3.3a was constructed on prototype board, with RS232 communication connected to laptop (COM1, 1000Mhz AMD processor, 256Mhz SDRAM, 8MB 3D accelerator card, running windows XP).

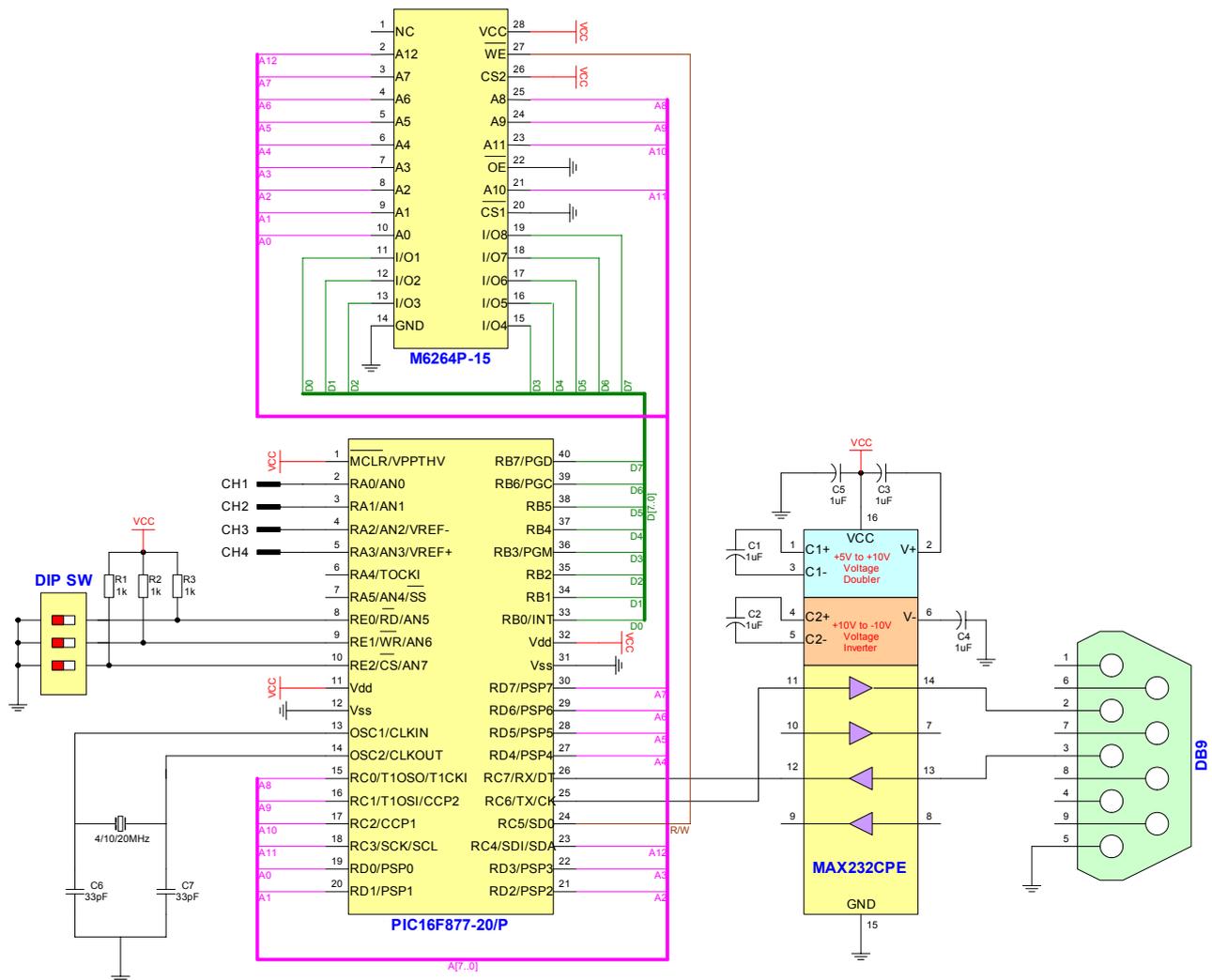


Figure 11.3.3a. Test Circuit Diagram 3

Program mark10.c was compiled and loaded into the PIC. This program carries out three tests: -

1. Fill all memory locations with 0xFFh and check.
2. Fill all memory locations with 0x00h and check.
3. Fill all memory locations with the LSB of the address and check.

RS232 communication is used to relay the results to the PC; the DOS based program test.exe used in RX terminal mode was used to receive and display the results.

Screen dump of the result is shown below: -

```

RX Terminal
-----
Waiting for message (press a key to stop)
TEST RAM CHIP...

Test 1 - Fill RAM with 0xFF ---> Passes = 7806, Fails = 386
Test 2 - Fill RAM with 0x00 ---> Passes = 8114, Fails = 78
Test 3 - Fill RAM with LSB of Address ---> Passes = 31, Fails = 8161

End of RAMTEST, Hopefully work can now start on storage mode
    
```

Clearly this test was a failure, it was discovered that there was a bug in the PIC program code (version 1.1). Recall that the DIP switches are connected to port E and in order to configure this port for TTL inputs, parallel slave mode is turn ON. It is clear from the PIC data sheet that this parallel slave mode also effects port D, which is connected to the address bus. Switching off parallel slave mode solved the problem, but created a new one, DIP switches cannot be read. The solution to this new problem was simple, before reading the DIP switches turn ON parallel slave mode, and after reading the DIP switches turn OFF parallel slave mode.

Screen dump of the result for the corrected mark10.c program is shown below: -

```
RX Terminal
-----
Waiting for message (press a key to stop)
TEST RAM CHIP...

Test 1 - Fill RAM with 0xFF ---> Passes = 8192, Fails = 0
Test 2 - Fill RAM with 0x00 ---> Passes = 8192, Fails = 0
Test 3 - Fill RAM with LSB of Address ---> Passes = 8192, Fails = 0

End of RAMTEST, Hopefully work can now start on storage mode.
```

Clearly the RAM test was a success, and work can now start on the storage mode embedded program.

11.3.4. Test 11: 5V Regulator

The system powering circuit (figure 5.5a) was constructed on prototype board. The input voltage was varied from 8 to 20 VDC, while the output voltage was measured using a digital CRT storage oscilloscope. Output voltage was 5 VDC, throughout the test.

11.3.5. Test 12: Test 12: Analogue Circuitry

The analogue circuitry (figure 5.4a) circuit was constructed on proto-type board and tested. Signal generator was used to generate the input, and a digital CRT storage oscilloscope was used to check the output before connecting to the PIC ADC. The test was not perfect, as the scale was incorrect. For example a -2.5 to 2.5 volt sine-wave was connected to the -2.5 to 2.5 input, the output was 1 to 5 volts and not 0 to 5 volts as expected. Allow this could be software corrected; it makes since to play around with the resistor values to try and improve this. (Perhaps variable resistors may have to be used for calibration purposes).

Note a switch mode DC to DC converter was used to converter 5 Vdc to -15 and 15V. Hence the entire circuit was power from one supply. Note the diodes successfully stopped the output voltage getting too low or too high.

11.3.6. Conclusion

Evidently the scope program can view up to four channels at a sample rate of 1kHz and one channel at a sample rat of 5kHz in real-time. Clearly there is a triggering problem as it was difficult to obtain a stable trace when triggering off another channel, for example when triggering off channel 1, all other channels were constantly scrolling across the screen. When the same signal is inputted into all four channels, it is now possible to get a stable trigger (e.g. waveforms related), but waveforms of different frequencies was a problem.

Clearly storage mode operation is now possible, as the RAM chip was successfully tested. The first step might be to use the real-time frame structure, storing up to 4000 reading into RAM and then cycling through them transmitting them to the PC in one large block. This is a quick and easy test, next step is actually writing the storage mode software and then the control protocol.

12.0. CONCLUSIONS

Clearly commercial oscilloscopes are over priced; the main goal of this project was that the cost of the PC based oscilloscope should be as low as practically possible, the total cost was expected to be less than £50 and this figure was used as a guide when selecting hardware components. Clearly this goal has been achieved as the total cost of PCB components was calculated to be £22.97 (retail price), obviously this does not include the cost of the PCB board (about £7), case (about £5) and manufacturing costs (about £15), but the total product costs should be under £50.

Note the software application should be freeware, including full source code; hence customers can modify code making improvements and adding additional features, these customers could be encouraged to send their customised copy of the program to the author so that some of these changes could be included in new releases of the program.

Clearly this project is cheaper than any other similar product on the market as the nearest was a 20 kHz single channel PC based storage oscilloscope (-5 to 5V) at £95. It is also clear that this project has many technical advantages over existing products, for example it is a quad channel oscilloscope and commercial quad channel oscilloscopes are extremely expensive (including commercial PC based scopes). It is foreseen that the oscilloscope could be available in kit form, were customers would buy the PCB components, PCB board, case, leads and build the scope themselves; this is a real option as the hardware circuitry is extremely simple and this would reduce product costs making the product more economical.

Core objective "design and construct the hardware required for data acquisition", clearly this objective has been successfully achieved. The PIC16F877 is used, this microcontroller is extremely powerful running at a 20MHz, it has a built in 10-bit ADC which can be connected to any one of 8 input pins, and a hardware UART suitable for RS232 communications.

Evidently the analogue circuit was proven to work, as it was simulated in electronic workbench and then in the laboratory using real components. Obviously the analogue circuit needs to be redesigned so that the PIC microcontroller can control the input voltage range, as stated in this report, there are a couple of different ways of achieving this. The first, analogue switches could be used to switch in different resistor values to the op-amps, hence changing the gain (e.g. MAX4066 or MM74HC4066). The second, and more likely option is the use of digital potentiometers (resistance controlled by microcontroller), to change the gain of the op-amps.

The circuit has built-in protection (diodes), to make sure the ADC does not get damaged by high or negative voltages, but there is no protection at the inputs to the op-amps. Clearly this is a problem, what happens if the wrong range is selected and 200V is fed into the -2.5V to 2.5V range, this will destroy the op-amps; allow the diodes at the PIC ADC pin should protect the PIC, hence it is inexpensive to fix. It is also important to note that resistors normally have a maximum voltage rating of about 200V, hence for safety reasons it is better for the 100kΩ resistor for the -250V to 250V input range to be made up of several resistors in series (i.e. 3 x 33kΩ, 1 x 1kΩ).

The RS232 transport medium was chosen, the main reasons why it was chosen is because it is easy to program, reliable and every PC has at least one RS232 port. The main disadvantage of RS232 is that it is slow (115Kbps max and will limit the maximum real-time sample rate) when compared to other mediums (e.g. USB 12Mbps). The USB port is ideal for this type of application, but because a USB port can have many devices daisy chained, the communication protocol must be compliant with the USB standard. This USB communication protocol standard is quite large and complex, and it was decided that it would take too much time to implement. Note the scope program is designed to be flexible, hence sometime in the future the transport medium could be changed allowing for faster real-time sample rates.

Included in this report were design details of the real-time communication protocol, this included the frame structure and some maximum sample rate calculations at different baud rates. There are two real-time sampling modes, the first is chop mode where channel 1 is sampled then channel 2 then channel 3 then channel 4. Clearly this causes a number of problems, the first is that RS232 throughput must be shared between all channels hence this reduces the maximum sampling rate, the second is that because it takes time for the ADCs sampling capacitor to charge (20µs), there is a time delay between sampling each channel (e.g. channel 1 will lead channel 2 by 20µs). Obviously this waveform offset can be software corrected, for

example a future version of the scope program could have a 'calibration' dialog which allows the user to software offset channel timings to compensate for this 20µs delay between waveforms.

The second real-time mode is 'alternate' basically this samples channel 1 1000 times, then channel 2 1000 times, then channel 3 1000 times, then channel 4 1000 times. Clearly this has the advantage of maximising the real-time sample rate (channels don't share RS232 throughput) no matter how many channels are enabled, but continuous monitoring of a waveform is impossible as there are gap in the data and channel phases are lost (e.g. CH1 leads CH2 by 90°). Allow since the PIC knows how much time has elapsed (PIC timer) since sampling the pervious channel, this information could be transmitted to the PC, where the scope program could software correct the phase of each waveform. Obviously this mode is only useful for periodical waveforms (e.g. sinewaves) and not for example an ECG signal.

Evidently the maximum real-time sample rate is 5.75 kHz, which is the maximum throughput of the real-time frame structure through RS232 at the maximum baud rate (115kbaud). Clearly there is no reason why more than one serial link could not be used to increase this real-time sample rate, for example two bytes are sent per ADC reading, hence COM1 could be used to transmit the first byte and COM2 could be used to transmit the second byte (11.5 kHz sample rate). Obviously the chosen PIC has only one hardware UART, hence a software UART is required for the second com port (easy to achieve using CCS compiler) but this will push the PIC to the limits but should be ok at 20MHz. No extra hardware (i.e. no extra cost), except for another serial cable is required as the MAX232 is a dual line driver chip hence there is a spare set of drivers that can be used for this additional com port. Undoubtedly if the PIC struggles to cope with the additional software UART, a separate hardware UART could be used.

Since the RS232 communications is the bottleneck when operating in the real-time mode, it is important that the real-time frame structure is as efficient as possible. Unmistakably that's the reason why the real-time structure does not have a check sum or CRC. Clearly, this causes a problem, however the frame structure is designed in such a way that some errors can be detected, but the cable between the MAX232 buffer and the PC will be high quality shielded cable (short length) and errors should not occur that frequently anyway. If one or two readings are lost/corrupted, it is clear the appearance of the waveform will not be affect too much, as there are at least 10 readings per cycle and because it is real-time the screen is continuously refreshed, hence the error will probably not stay on the screen for very long until it is replaced by new readings.

Clearly a Windows based C++ application capable of displaying low-frequency waveforms from four channels simultaneously in real-time has been successfully designed, implemented and proven. The scope program graphically displays waveforms without flicker, it has a good user interface that is easy to use, and directly communicates with the PIC. There were many problems during development, some of which have been described in detail in this report (e.g. Windows 95/98/ME compliably problem). These problems are all part of the learning curve, for example one of the early problems was drawing the traces without flicker, and this problem was solved by drawing the trace to a memory DC (bitmap) and then quickly coping this to the screen. Before hand it was not known how to achieve this, hence new knowledge has been acquired. Clearly the graphical display works well and is extremely flexible (e.g. auto resizable, user adjustable grid resolution and colour).

The graphical display is designed to display the waveforms as it would appear on a traditional oscilloscope. The standard resolution (allow this can be modified up to 99 by 99) is 10 by 8 squares (that's 100 by 80 dots), and has a user configurable trigger (software trigger). This triggering operation took some time to implement, there were many problems at first, but eventually most of the problems were ironed out, a view still remain e.g. unstable CH2 trace when triggering off CH1 if waveforms are unrelated. The user can select from a range of different triggering methods (e.g. low to high, high to low, level or edged etc...) and which channel they which to trigger on (e.g. 1,2,3, 4), including impendent triggering where each channel is trigger independently of each other, this feature is not possible on traditional analogue oscilloscopes, it is not sure how useful this feature is, buts its another one of those features ('Bells and Whistles') that gives this project an edge over existing similar products. Notice the split screen feature; were the graphical display is on the left hand side and the controls are on the right, just as they would appear on a traditional oscilloscope.

It is clear that the scope program can display multiple scope windows, each one with its own selected channels enabled, impendent triggering, time-bases, scales and grid resolution. It was not sure how useful this feature might be and during development it was considered that this feature may have been dropped on the finished application. Fortunately this was not the case and indisputably it may have some use for specialist applications and it's another one of those features ('Bells and Whistles') that sets this project apart

from other similar products. It is clear that there is no limit on how many scope windows can be opened simultaneously, perhaps until the computer runs out of memory, which on a modern PC could be 1000s of windows.

Evidently direct access to the hardware is not allowed under Windows NT/2000/XP. Hence the reason why interaction with the serial port was achieved using a file handle and various WIN32 communication API's. Note this method is also Windows 95/98/ME compatible.

The scope program has been successfully tested on Windows 98/NT/2000/XP and there is no reason why the program should not work on Windows 95/ME, allow comprehensive testing is required. Simulated random noise tests running for hours non-stop, was used to test the reliability of the scope application, (e.g. can it recover when the noise is removed or has it crashed) the program continued to operate correctly once the noise was removed, but when the noise was present (no CRC check in real-time mode) all sorts of strange waveforms were being drawn on the display.

Clearly there is a problem when the user has selected an out of range time-base (i.e. aliasing can occur, even if sampling at more than twice the waveform frequency), recall that the scope program increases the time-base by skipping readings and decreases the time-base by duplicating readings. Obviously it is possible for aliasing to occur (skipping reading) if the selected time-base requires for less than 2 samples per cycle to be displayed, hence it is important that the user makes sure that the selected time-base is optimal for the waveform being monitored at the selected sample rate.

Evidently the scope program is far too processor hungry and future modification is required to make it more efficient, for example the trigger point for all enabled channels are always calculated even if all channels are triggering of one channel, clearly this is inefficient and bad programming practice and needs to be improved. It has been proven that the scope program requires a reasonably fast PC to run smoothly, the minimum spec being 200MHz (for a reasonable refresh rate) and probably 500MHz for quad channel operation at a high refresh rate (50ms) and resolution. Clearly modern PCs will have no problems running the scope program as the minimum spec computer on the market today are all above 1GHz with the top-range above 2GHz, and computers up to 3 years (300 MHz or above) old should comfortably run the scope program.

Obviously the development of the scope program is not complete and there remains many additional features yet to be added, for example the addition of 4 user configurable virtual channels (e.g. VC1 = CH1 + CH2, or CH1 - CH2, or CH1 * CH2, etc...), auto frequency calculation, auto time-base, auto voltage scale, print directly from scope program and storage mode operation. Overall good progress was made, the program can display up to four real-time waveforms either scrolling across the screen or triggering, waveforms can be saved and reopened, graphical display can be exported to a bitmap or the Windows clipboard, voltage scales are adjustable, zero point and waveform offset easily adjustable using unique user interface (small triangles) and channels are easily disabled/enabled.

The philosophy used during the development of the PIC embedded software was to keep it simple, straightforward, comprehensible, and to a minimum. Many small programs were designed for testing the hardware and ideas; clearly the end result is that the PIC code was gradually built up step-by-step, instead of writing the entire program at once. Evidently this ensures that operational results are obtained, as testing procedures were carried out at each stage, while if the program had been written all at once, there would have been little chance of it working and could have proved difficult to debug. The high-level programming language C was chosen and not the low-level assembly code normally associated with PIC programming, clearly there are many advantages of this including: ease of programming, ease of modification, reusability of code, and use of standard functions. Clearly PIC embedded software (mark9.c) was successfully designed, implemented and tested to read the ADC at a certain sample rate (interrupt time-base) and transfer the data to the PC using the real-time communication protocol.

It is clear that a Windows dialog based simulator program was successfully developed, this program is easy to use: the user selects the type of waveform, amplitude and frequency for each channel. The simulated waveforms are then transmitted to the scope program using the real-time frame structure (storage not implemented). Clearly accurate timing is difficult to achieve, and no attempt has been made to achieve this as it would have wasted a large amount of time, plus the timing of the simulator would have been completely different to the timing of the PIC hence the code would not be reusable (actual timing on PIC only), clearly even though the simulator does not have actual timing it still is a powerful tool, it will allow the communication protocol, graphical display and triggering method of the scope program to be easily tested

before the hardware is constructed and PIC embedded software is written. This program works well and was successfully used to test the scope program, allow storage mode operation and the control protocol are yet to be implemented; hence more development is still required.

Evidently the entire system has been demonstrated to work together as a basic real-time low-frequency oscilloscope (up to 5 kHz) as the test results show in chapter 11. Clearly the maximum possible real-time sample rate is 5.75 kHz (with one RS232 link) a low pass filter is required (anti-aliasing) to filter all frequencies above half the sample rate, in this case the filter should filter out frequencies above 2.875kHz, hence because the system has a low frequency cut off of 0Hz (e.g. DC waveforms can be monitored) the system has an analogue bandwidth of 2.875 kHz.

Clearly there is no reason why multiple scopes cannot be used, for example two scope programs can be executed in windows one could be configured to use COM1, the other COM2. Hardware can be connected to COM1, and other set of hardware connected to COM2. Hence up to 8 channels can be monitored simultaneously on one computer screen. Obviously the only limit is the number of serial ports the PC has and how many scope programs it is capable of running simultaneously (i.e. processor power). Also it is important to note that the chosen PIC has 8 input pins to the ADC, hence with minor modifications to the scope program (cut & paste drawing of waveforms, etc...), hardware (remove dip switches from port E), and embedded software (sample channel 1 to 8) eight channels could be monitored simultaneously (that's 16 channels if two scopes are used), but at a reduced sample rate when all eight channels are enabled (unless using storage or real-time alternate mode).

Adding storage functionally to the scope was the last thing to be done, clearly this feature is not complete but good progress has been made as block diagrams of the communication protocol and frame structure were included in this report. It is important to rely that storage mode, should theoretically allow for faster sample rates which are not limited by the speed of the transport medium; e.g. the PIC reads 1000 readings from channel 1 at 20KHz then stops sampling and transfers the data to the PC using the storage mode frame structure (including CRC) via RS232, once ACK (Acknowledgement) is received from the PC the PIC starts sampling again. If NAK (Negative Acknowledgement) is received or a timeout occurs the PIC retransmits the frame. Clearly most of the ground work for implementing storage mode operation has been done, for example functions for writing to the RAM chip were successfully written and tested, and implementation can easily be achieved if more time was available.

Evidently the maximum sample rate when operating in storage mode is the speed of the ADC, this was calculated to be 43.63 kHz. Hence since frequencies above half the sample rate must be filtered (anti-aliasing) the actual maximum analogue bandwidth of the system is 21.815Hz (note $f_{low} = 0\text{Hz}$). Undoubtedly an external ADC with direct memory access makes it possible to monitor very high frequencies (e.g. AD9070 – 10bit, 100MSPS ADC), but these chips are expensive, hence the low cost of the design is lost. It is clear the following limits the maximum storage sampling rate: the maximum sampling speed of the ADC, time taken to send to RAM, and the amount of memory.

Overall progress has been good, although it has taken much longer than expected to complete certain aspects of the project (e.g. triggering of waveforms). The project was a success, real-time waveforms have been successfully displayed on a PC in real-time and it has been demonstrated how storage mode operation could be added to project to increase maximum sampling rate, even if the transport medium is slower than the sample rate.

13.0. RECOMMENDATIONS FOR FURTHER WORK

Clearly there is a lot of additional development work that can be done to improve the design of this low cost PC-based quad channel real-time / storage oscilloscope. This chapter will briefly highlight areas of the design that merits improvement and additional features to make the scope more powerful, reliable, flexible, and commercially marketability.

13.1. Control Protocol

Obviously the first thing that needs to be finished, is the control protocol allowing for the scope program to directly control the sample rate, enable / disable channels, real-time / storage mode, chop or alternate, etc... Sufficient details have been included in this report on how this might be achieved.

13.2. Storage Mode

Sample a finite number of samples storing them into RAM. Once the memory is full (or the preset number of samples has been reached) the PIC will stop sampling and transfer the data to the PC, when ACK (acknowledgement) is received from the PC the PIC will start sampling again. There is enough information contained in this report to implement this feature, clearly the scope program and PIC software needs modification.

13.3. Multiple Serial Ports

The PIC only has one hardware UART; hence one solution is to use the hardware UART plus a software UART (CCS C complier makes this easy). Two bytes are sent for per ADC reading, hence put the first byte on COM1 and the second on COM2, this will double the maximum real-time sampling rate (11.5 kHz max). Note the MAX232CPE chip is a dual RS232 line driver chip hence no extra hardware is required, allow the additional software UART will push the PIC to the limit (should be OK at 20MHz). If it turns out it is not practical to use a software UART, an additional hardware UART chip could be used, these chips are readily available and are reasonably priced.

13.4. USB Transport Medium

The USB port is ideal for this type of application (12 Mbps); this takes away the RS232 bottleneck providing faster real-time sample rates. Note the USB throughput is far faster than the PICs ADC (43.63 ksps) hence this is the new bottleneck of the system. There is no need for storage mode, as the USB throughput is capable of transmitting a sample rate up to 600 kHz (12,000,000/20) in real-time, using external ADC.

Note the USB port can have many devices daisy chained; the communication protocol must be compliant with the USB standard. This USB communication protocol standard is quite large and complex, hence a dedicated PIC maybe required to deal with USB communication, while another does the sampling. Note that USB communication protocol is packet based (lets other devices share the same link) hence the actual throughput will be less than stated above. Note the scope program is designed to flexible, and should be easy to integrated with the USB standard, actually Microsoft probably has a standard C++ plug-in object available for visual C++ designed for letting C++ programmers (e.g. the scope program) use the USB port without total knowledge of the USB communication protocol (e.g. automatically converts into correct packets). Unfortunately interfacing the PIC will be a completely different story.

13.5. Bootstrap

The chosen PIC can protect blocks of program memory, hence a bootstrap program can be written to check the serial interface before calling the main program. If a certain block of characters are received during bootstrap, it moves into program mode, using a bidirectional communications protocol (RTS, ACK, NAG,

etc...) a new program can be download from the PC directly into program memory, when download is complete the new program is called. This offers greater product flexible as software updates (bug fixes, new features) can be downloaded free of charge and download directly into the product using the same serial interface and software used to display the waveforms (i.e. the scope program).

Microchip application note [J6] will fully describe how to implement this, including full source code.

13.6. Virtual Channels

It is recommended that the scope program should have up to four user configurable virtual channels, that's a total of 8 channels. Each virtual channel can be configured as follows: CH1 + CH2 or CH1 x CH2 or CH1 – CH2 or CH1 / CH2 etc... This makes the scope program more versatile increasing its usefulness, for example monitoring of RS485 / RS422 serial communication is now possible. RS485 / RS422 works on the principal of differential voltages between two cables twisted together (good noise immunity), hence two oscilloscope channels (four for RS422) could be used and the scope program will automatically add the traces together creating a new trace say CH5 (virtual channel 1).

This is extremely easy to achieve, it is only matter of creating four more traces the same as was done for channels 1 to 4 (e.g. cut & paste makes life easy). Instead of filling the arrays for these traces using the communication protocol, the program continuously scans though each real channel array multiply, adding, dividing or subtracting channels together (e.g. whatever the user has specified for each virtual trace) filling the virtual trace arrays.

13.7. Data Logging

The scope program already has dummy dialog controls for this feature, basically real-time data is streamed to disk, allowing the user to play back the recorded data in real-time, manually scroll through the data, and email the data to a colleague anywhere in world. For example this feature would be useful for monitoring of an ECG signal, where the raw data can be email to a doctor for diagnosis.

This feature is easy to implement basically every character that is received from the serial port is dumped into a file, and when playing back the data, the program runs through the recorded data at the same rate it was originally sampled at (perhaps the sample rate combo box could still be operational, allowing for a slowed or speeded up view of the data).

13.8. TCP / IP Internet Communications

Remote monitoring / control of oscilloscope from anywhere in world, for example the machine actually connected to the scope hardware (running in slave mode) relays real-time data through the internet using the TCP / IP internet protocol, the scope program on the other end (master program), receives the data and draws the waveforms as normal. The scope program operating in slave mode has its controls disabled, hence only the master scope program controls the PIC, via the internet and via the slave scope program.

Problem the internet (unless using broadband technology, e.g. ASDL, fibre optic) is slow and commonly suffers from net congestion, hence this feature is only useful for slow sampling rates, unless storage mode is used. For example an ECG signal only requires a sample rate of 120 Hz, clearly the internet is capable of transferring this data in real-time without any problems, hence a doctor can remotely diagnosis an ECG signal in real-time.

13.9. Direct Modem to Modem Communication

The master scope program dials the slave scope program and a direct communication link is established. Once at direct modem link is established the communication protocol between the remote scope programs are exactly the same as with a direct like to the PIC, hence the only operation of the slave program during this time is to relay the data from the PIC to the master scope program and visa versa.

This feature is easy to implement, basically two additional functions is required one to setup the scope program as a master (dial user configurable number, real-time, storage and control protocols unchanged) and the other to setup the scope program as slave (wait for connection and once connection is established rely data from the PIC to the master visa versa).

13.10. Printing of Scope Display

The scope program should be able to print the scope display directly to the system printer. Some work has already been done in this area, for example configuration of the printer is currently possible and code for printing exists along with a header and footer. The only thing that remains to be completed is the drawing of the scope display, clearly the simplest way of achieving this is by copying the bitmap directly to the printer output CDC, but this is not ideal, as this would not contain any information about the selected time-base, voltage scales (this is essential), and the screen resolution (source of bitmap) could be much lower (this is likely) than that of the printer hence producing a poor quality printout.

Obviously the grid and traces need to be redrawn on the printer output CDC, this is simple as the code is exactly the same as that which draws the grid and traces to the screen (hence put cut & paste to good use), allow the size of the page is different to that of the screen hence don't forget to detect the resolution of the printer.

13.11. An Alternative Method for Dealing with Aliasing

Generally a low-pass filter is used to filter out frequencies above half the sample rate. Instead of filtering out aliased signals, a hardware circuit can be used to detect the actual frequency of the waveform being monitored. The PIC can then transmit the actual frequency of the waveform to the scope program. The scope program can then software calculate the frequency of the sampled waveform and if the two frequencies do not match the scope program knows that the sampled waveform is an alias.

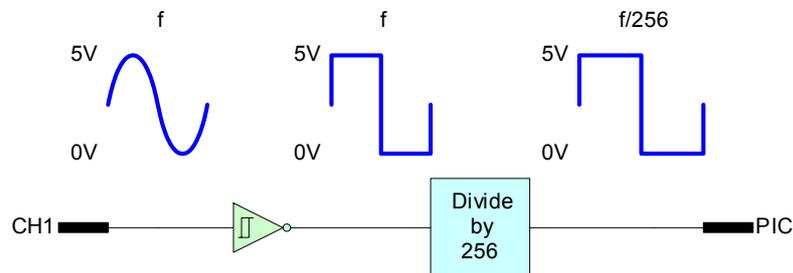


Figure 13.11a. Frequency detection circuit

Figure 13.11a shows how a simple detection circuit could be made. A Schmitt-Triggered inverter is used to generate squarewaves at the same frequency of the original signal which is then passed through a frequency divider circuit which decreases the frequency by 256. The output of the detection circuit is connected directly to a Schmitt-triggered input on the PIC which is configured to interrupt the PIC. Using a real-time timer the frequency of the signal can be calculated and transmitted to the scope program.

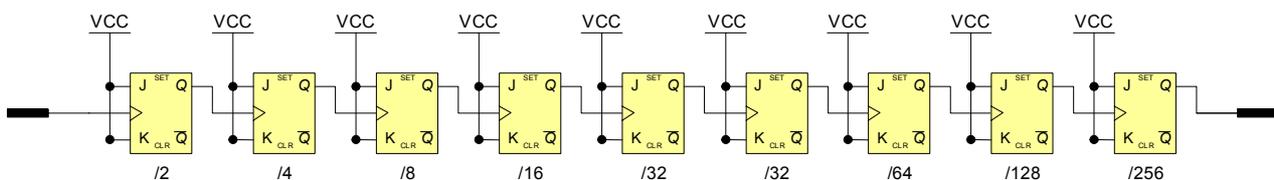


Figure 13.11b. Divide by 256 circuit

The reason why the divider circuit (figure 13.11b) is required is to reduce the processing load on the PIC. For example if the divide by 256 was not there, the PIC would be interrupted 5000 times a second instead of 19.5 for a 5 KHz signal. If the PIC was interrupted 5000 times a second, normal operation of the PIC would

be severally affected. If extremely large frequencies are possible ($> 1\text{MHz}$) the frequency divider should be increased.

13.12. Taking Advantage of Aliasing

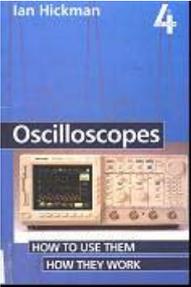
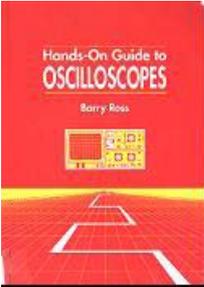
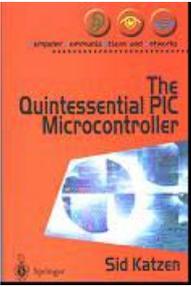
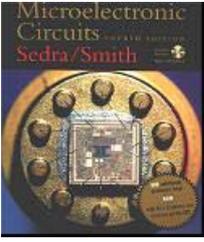
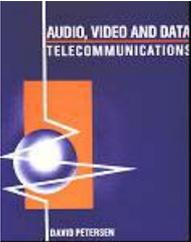
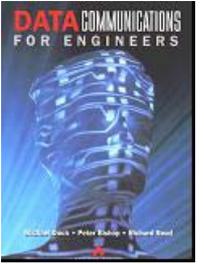
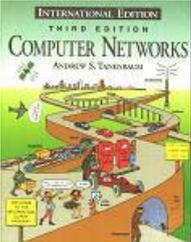
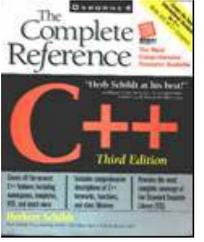
Using the frequency detection circuit shown in figure 13.12a it maybe possible to monitor periodical waveforms that have a frequency much higher than the sample rate. The idea behind this theory is that at certain frequencies aliased waveforms are the same as the input waveform expect for the frequency (as seen during testing). Therefore with some software processing it is possible to redraw the waveform at the correct time-base using the hardware frequency detection circuit as a guild. This technique will not work for random waveforms for example sound-waves, only periodical waveforms which 9 times out of 10 is what an engineer uses an oscilloscope to monitor.

This technique could allow for waveforms up to 1 MHz (or above) to be monitored using this PC based oscilloscope. If more time was available it would be nice to investigate this theory further.

14.0. BIBLIOGRAPHY / REFERENCES

NOTE: References are in blue text (referred to them explicitly in this report), the rest are bibliography.

Text Books

- | | | | |
|--|--|---|--|
| <p>[B1]</p>  | <p>Oscilloscopes – How to use them, How they Work</p> <p>By Ian Hickman
 Publisher: Newnes
 ISBN: 0-7506-2282-2
 UJLIB: 621.3815483HIC
 1995</p> | <p>[B2]</p>  | <p>Hands-On Guide to Oscilloscopes</p> <p>By Barry Ross
 Publisher: McGraw – Hill
 ISBN: 0-07-707818-7
 UJLIB:621.3815483ROS
 1994</p> |
| <p>[B3]</p>  | <p>The Quintessential PIC Microcontroller</p> <p>By Dr Sid Katzen
 Published by Springer
 ISBN: 1-85233- 309-X.
 2001</p> | <p>[B4]</p>  | <p>Microelectronic Circuits – Fourth Edition.</p> <p>By S. Sedra and Kenneth C. Smith
 Publisher: Oxford University Press
 ISBN: 0-19-511690-9
 1998</p> |
| <p>[B5]</p>  | <p>Audio, Video And Data Telecommunication</p> <p>By David Petersen
 Published by McGraw-Hill
 ISBN: 0-07-707427-0
 1992</p> | <p>[B6]</p>  | <p>Data Communications for Engineers</p> <p>By Michael Duck, Peter Bishop and Richard Read.
 Publisher: Addison – Wesley Longman limited.
 ISBN: 0-201-42788-5
 1997</p> |
| <p>[B7]</p>  | <p>Computer Networks – Third Edition</p> <p>By Andrew S. Tanenbaum
 Publisher: Prentice Hall
 ISBN: 0-13-394248-1
 1996</p> | <p>[B8]</p>  | <p>The Complete C++ Reference</p> <p>By Herbert Schildt
 Published by McGraw-Hill
 ISBN: 0-07-882476-1
 1998</p> |

Journals

- [J1]** “PC-based virtual test and measurement speed bench testing” by Ivor Matanle (AA Vol. 15 No. 4, 1995 - page 12)
- [J2]** “Digital signal processing“, by Andy McFarlane (Journal: Sensor Review, Year: 1997, Volume: 17 Number: 1 Page: 13 -- 20, Publisher: Emerald).
- [J3]** “Computer integrated documentation” by Robert M.Parkin and B.S. Dalay (Industrial Management & Data Systems, Vol. 94 No. 1, 1994, pp. 3-8).

- [J4] "Anti-aliasing, analogue filters for data acquisition systems", by Bonnie C. Baker (Microchip application note AN699, © 1999).
- [J5] "Using digital potentiometers to design low pass adjustable filters", by Bonnie C. Baker (Microchip application note AN737, © 2001).
- [J6] "Implementing a bootloader for the PIC16F87X", by Mike Garbutt" (Microchip application note AN732, © 2000).

Web Sites

- [W1]  <http://www.farnell.com> (Farnell Online catalogue).
- [W2]  <http://rswww.com> (RS Online Catalogue).
- [W3]  <http://www.microchip.com> - Microchip Website (PIC datasheets and application notes).
- [W4]  <http://www.codeproject.com> - The Code Project (a community of Windows developers specialising in C++, MFC, C# and .NET who have come together to share source code, articles and tutorials).
- [W5] <http://www.lvr.com/usb.htm> - USB Central (Information, tools, and links to material about the Universal Serial Bus).
- [W6] <http://www.lvr.com/serport.htm> - Serial Port Central (A collection of files and links to material relating to serial links and networks, especially in monitoring and control applications).
- [W7] <http://www.lvr.com/parport.htm> - Parallel Port Central (A collection of files and links to material about the PC's parallel port, including ECP, EPP, bidirectional, and IEEE-1284 modes).
- [W8]  <http://msdn.microsoft.com/> - Microsoft Developer Network
- [W9] <http://www.engj.ulst.ac.uk/sidk/PIC/index.html> - PIC Resource Site (An assorted set of data sheets, instruction sets and web sites, by Dr S Katzen).
- [W10]  <http://www.programmersheaven.com/> (Over 18449 resources to explore)
- [W11] <http://www.codeguru.com/> (C++ source code, etc...)
- [W12] http://www.intelinfo.com/newly_researched_free_training/C++.html (Free C++ Programming Training and Tutorials)

[W13] www.picotech.com (PC Based Test & Measurement)

Data Sheets

[D1] "PIC16F87X Data Sheet", 28/40-pin 8-bit CMOS flash microcontrollers (© 1999 microchip technology)

A. CD ROM

This CD ROM contains complete source code (include compiled *.exe files) for all PIC embedded software, the test program, the simulator program and the scope program (all 42 versions present). Also contained on this CD ROM are datasheets and over 200MB of raw research data.

