308

AN AUTOMATED HARMONIC ANALYZER FOR USE IN STUDYING THE COMMERCIAL POWER SYSTEM

by

NORMAN MORTENSEN

BSEE, Kansas State University, 1986

A MASTER'S THESIS

submitted in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

Gary Johnson
Major Professor

ACKNOWLEDGEMENTS

The work detailed in this document was funded by the Kansas Electric Utilities Research Program (KEURP).

I would like to thank my major Professor Dr. G. Johnson, especially for putting up with missed deadlines. Also, I would like to thank Dr. Al Heber and Dr. Anil Pahwa for serving on my graduate committee.

Lastly, I would like to thank my wife Pam for her patience, understanding and, last but not least, her typing efforts.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1 : INTRODUCTORY MATERIAL

Introduction

The increasing use of sensitive electronic equipment has led to the need for higher quality commercial power. The quality of power can be determined in a number of ways. One method would be to examine a power system for transient surges and other such phenomena. Another technique would be to investigate the steady state spectral purity of an electrical utility's voltages and currents. This latter method is the one used in this study.

In 1985 Dr. Gary Johnson proposed conducting a study of the harmonic content of the Kansas electrical grid to the Kansas Electric Utilities Research Program (KEURP). The problem in conducting this study was the lack of a cost effective harmonic analyzer with the appropriate capabilities on the commercial market. For this reason Dr. Johnson proposed the development of a suitable harmonic analyzer as part of the project. This proposal was accepted by KEURP.

The harmonic analyzer that emerged from this project is a portable self contained unit with all the capabilities needed to monitor power system harmonics. Its user interface has been simplified by making use of current trends in portable personal computers. This harmonic analyzer is described in depth in the remainder of this document.

A Brief Overview of Power System Harmonics

Ideally, commercial power should possess a frequency spectrum in which all the energy is concentrated at 60Hz. In practice some of the energy is associated with other frequencies. The purpose of this study is to measure this distribution of energy.

If a signal is a periodic waveform then it can be expressed as the sum of sinusoidal waveforms whose frequencies are the frequency of the waveform and its multiples. The frequencies which are multiples of the fundamental frequency of a waveform are called harmonics. Since the waveforms on a commercial power system are generally periodic most of the energy associated with a waveform will be concentrated at the fundamental frequency and its harmonics. The harmonic analyzer described in this document measures the harmonics of the voltage and current waveforms from which the distribution of energy can be calculated.

Harmonics are caused by a number of elements on the system such as generators, transformers, solid state motor controllers and inverters. Due to the fact that the windings of a generator are not perfectly aligned the generated voltage is not a perfect 60Hz sinusoid. Additional harmonics are caused by various nonlinearities of the system. Magnetic saturation in the generators and transformers accounts for part of this. Other inherently nonlinear devices such as motor speed controllers and inverters also generate harmonics on the power system.

The presence of harmonics on a utility can pose problems to both the utility and the consumer of commercial power. These problems include degraded metering accuracy, interference with the telephone system and the heating of motors and generators. If a circuit is resonant at a specific frequency the magnitude of the voltage and current in the circuit may reach damaging levels. Harmonics can also cause saddles in the speed torque curves of induction machines. Additionally, the wavelength of harmonics also becomes much shorter than the 60Hz wavelength. This may cause the power system to begin to exhibit transmission line effects at harmonic frequencies. These effects of harmonics make it beneficial to both the utility and the consumer of commercial power to have a low harmonic content on the power system.

CHAPTER 2 : HARMONIC ANALYZER HARDWARE

Introduction

The harmonic analyzer described in this document was designed to meet a number of criteria needed to make it a practical device for measuring power system harmonics. These include the ability to measure numerous harmonics at once, independent operation, long term recording capability and power failure protection. Other features that would be desirable are portability, low power consumption and ease of use. These requirements point to a special purpose data acquisition system with onboard data processing.

A data acquisition system such as this would require a fast, easy to use processor for signal processing and control. For this reason the harmonic analyzer was built around the Motorola 68000 microprocessor. The 68000 is a high performance, versatile processor that is easy to use. It provides the speed and computing power necessary for signal processing work. The 68000 is also simple to program and ideal for use as an imbedded controller in an independently operating system.

The resulting system provides the features of a data acquisition system coupled with some of the signal processing capabilities of a general purpose computer. Some of the system's features include:

* 64 KBytes of RAM storage for use as scratch pad and data storage.
* 16 KBytes of ROM for control program storage.
* Real Time Clock to monitor data collection times.
* Sampling ports for the current and voltage waveforms designed to minimize the influence on the system.
* Data protection in the event of a power failure.
* Two serial ports for external communications.

A block diagram of the harmonic analyzer is shown in Figure 2.1. It shows the basic subsystems of the harmonic analyzer.

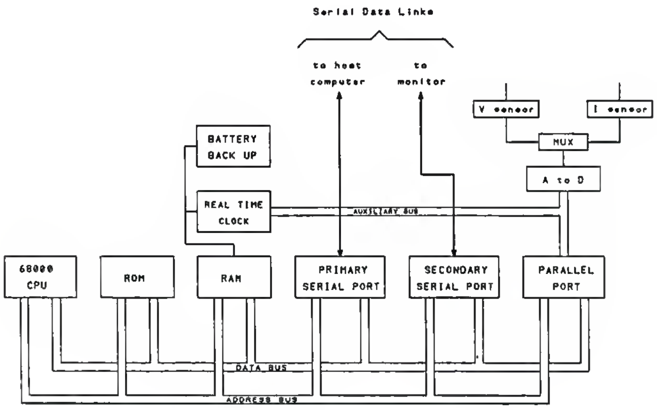


Figure 2.1 : Block diagram showing the basic subsystems of the harmonic analyzer.

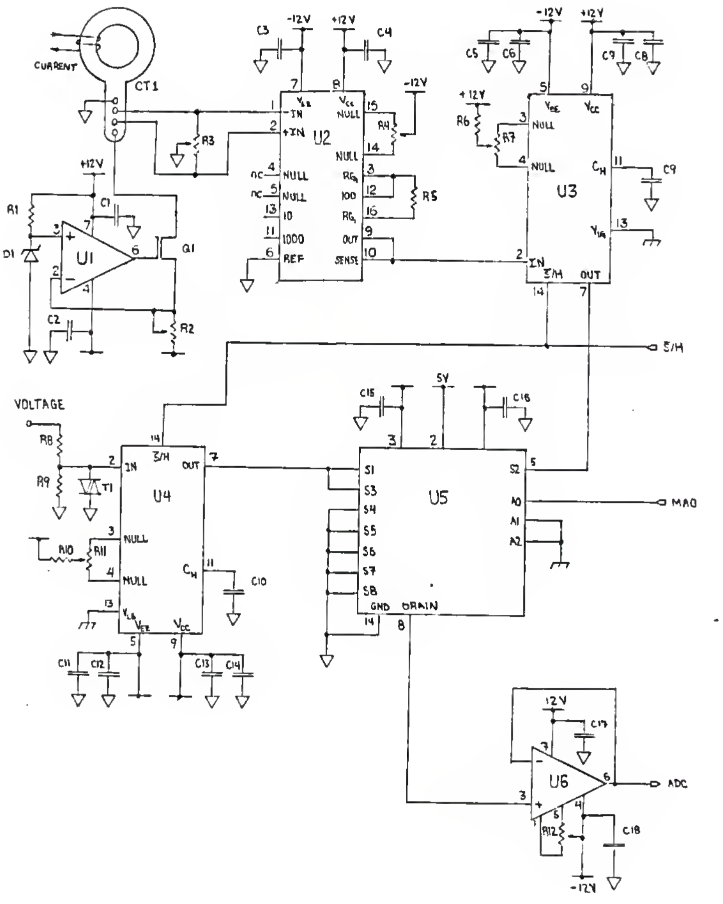
Analog Section

The analog section of the harmonic analyzer is made up of the voltage and current sensors and related circuitry. The schematics of this portion of the harmonic analyzer are shown in Figure 2.2. The list of parts is given in Appendix A. These sensors convert the line level voltages and currents to analog voltages in the +/- 10 volt range. An analog to digital conversion is then performed on these voltages and the rest of the signal processing is done digitally.

As can be seen from the schematics no antialiasing filter is included to process the signal before it is digitized. This was decided upon for a number of reasons. The amount of energy expected to be found at frequencies that would be aliased was expected to be small based on available information. The aliasing of these frequencies should then cause little error in the measurements. A filter with very steep skirts would have been necessary to properly filter the signal. This would have caused phase distortion in the signals and taken up extra board space. For these reasons the antialiasing filter was omitted.

The current sensor CT1 is a hall effect current sensor. It produces a differential output voltage proportional to the current that flows through it. The voltage is also

Figure 2.2 : Analog Circuitry



proportional to an excitation current used to drive the device. Since the voltage produced by an expected current of 5A is very small (on the order of 3-5 mV) the conductor carrying the current is coiled so that it passes though the sensor 10 times. This increases the output voltage from the sensor by an order of magnitude. The sensor itself is seen as a short by the system since the current conductor only carries the current through the sensor. This should lessen the effect of the analyzer on the system that it is measuring.

The excitation current for the current sensor is provided by opamp U1 and MOSFET Q1. These form an almost ideal current source. The opamp is setup in a negative feedback loop to maintain the gate voltage of Q1 so that the voltage across R2 is equal to the voltage across the reference diode D1. The amount of current flowing through Q1 can then be controlled by the resistance of R2, a multiturn trimmer potentiometer. Since the output voltages on the current sensor will be between the voltages on the current supply terminals the current source was designed to work between ground reference and the negative supply. This causes the outputs of the sensor to be close to ground.

The output voltages produced by the current sensor are amplified to the desired signal level by U2, a precision instrumentation amplifier. The trimmer resistor R3 across the outputs of the current sensor serves two purposes. It provides the bias currents for the inputs of the instrumentation amplifier. It can also be used to null the offset produced by the current sensor. The instrumentation amplifier is set up for a gain of 200. This was done by placing R5 in parallel with one of the internal resistors of the instrumentation amplifier. This configuration provides better temperature stability.

The signal flows into a sample and hold amplifier, U3. The hold capacitor C9 is a 5000 pF polypropolene capacitor. This was chosen for its low dielectric absorption and its temperature stability. A value of 5000 pF was chosen since the sample and hold is optimized for best performance with this value of a hold capacitor. The sampling control signal S/H is obtained from the timing and control circuitry discussed in the next section.

The voltage sensor is made up of a resistive divider that converts the 120 Vrms line voltages to voltages that lie within the +/- 10V range. The input impedance of the voltage sensor is 100K. This should produce little effect on the system being monitored. A transorb T1 protects the input of the sample and hold amplifier. The sample and hold, U4, is configured the same for the voltage sensor as for the current sensor.

The signals from the sample and holds for the current and voltage sensors are then routed through U5, an 8 channel multiplexer. The address line A0 is controlled by the signal MA0 which is generated by the PIA (U40, Figure 2.8) under program control. This determines which sensor signal flows through to the ADC.

The output of the multiplexer is buffered by U6, a voltage follower. This is an opamp specifically designed for voltage follower applications. Its output impedence is very low, less than 5 ohms. This is needed to drive the analog to digital converter since it has a dynamically changing input impedence during a conversion. A large enough impedence on the driving element could cause conversion errors.

The signal then flows into the analog to digital converter to be quantized. The remainder of the signal conditioning and processing is done in software.

Digital Section

The digital section of the harmonic analyzer makes the bulk of its circuitry. It performs the communication, storage, signal processing and general purpose control tasks of the harmonic analyzer.

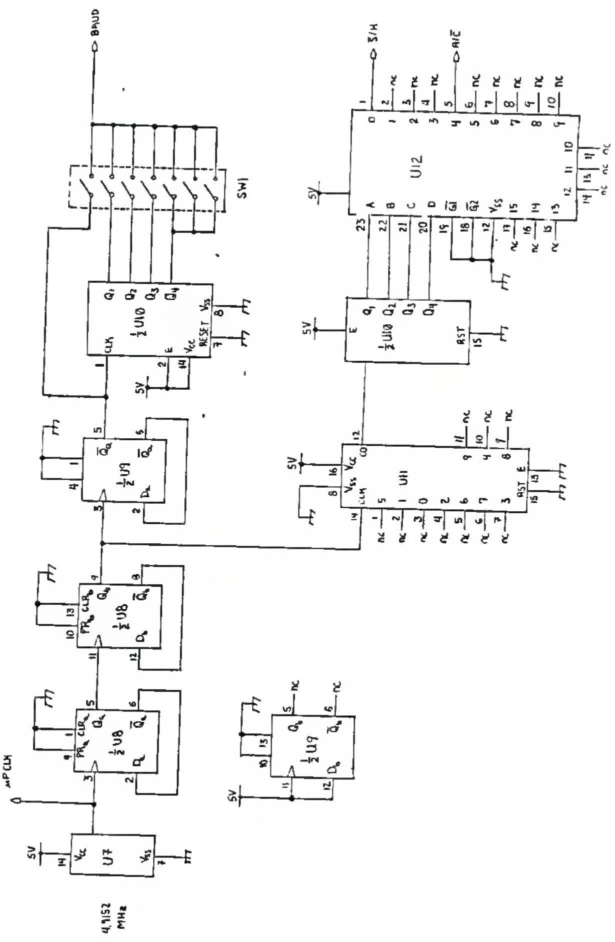
Signal Description Standards:

The signal description standards used in this document will be as follows. An active low signal will be referred to with an asterisk preceding it, e.g. *VPA, *SM1. A signal not preceded by an asterisk is assumed to be active high. Certain signals whose state determines function, such as the read-write line, will be written so that the state to cause a function is marked using a bar to symbolize active low. No symbol is needed to symbolize active high for a state. The read-write line is written as $R/\overline{W}$. Since many of the signals in this system are active low, a signal will be said to be active or inactive instead of high or low in order to make this discussion easier to follow.

Timing and Controls:

Figure 2.3 shows the circuitry for generating the various timing and control pulses used in the harmonic analyzer. These include the uP clock, sampling control and

Figure 2.3 : Timing and Control Circuitry



rate generation.

Timing for the entire system is derived from a 4.9152 MHz clock module, U7. This is used directly for the microprocessor clock. The 4.9152 MHz clock is then divided by 4 by the D flip-flops in U8. This results in a 1.2288 MHz clock signal.

The baud rate is generated by further dividing the frequency by two with half of U9. The resulting 614.4 KHz signal is then 9600 baud x 64. U10 generates the other baud rates by successively dividing the signal by two to get 4800x64, 2400x64, 1200x64 and 600x64. The frequency used is selected by SW1.

To generate the timing for the data sampling operation the 1.2288 MHz signal is divided by 10 using U11, a sequencer/divide by ten chip. This signal then drives the other half of U10, a 16 state counter whose output in turn drives a 16 state decoder, U12. The sample control S/H is the state 0 output line and the conversion control R/C is the state 5 output line. These signals provide all the timing needed by the various systems of the harmonic analyzer.

Microprocessor and Support:

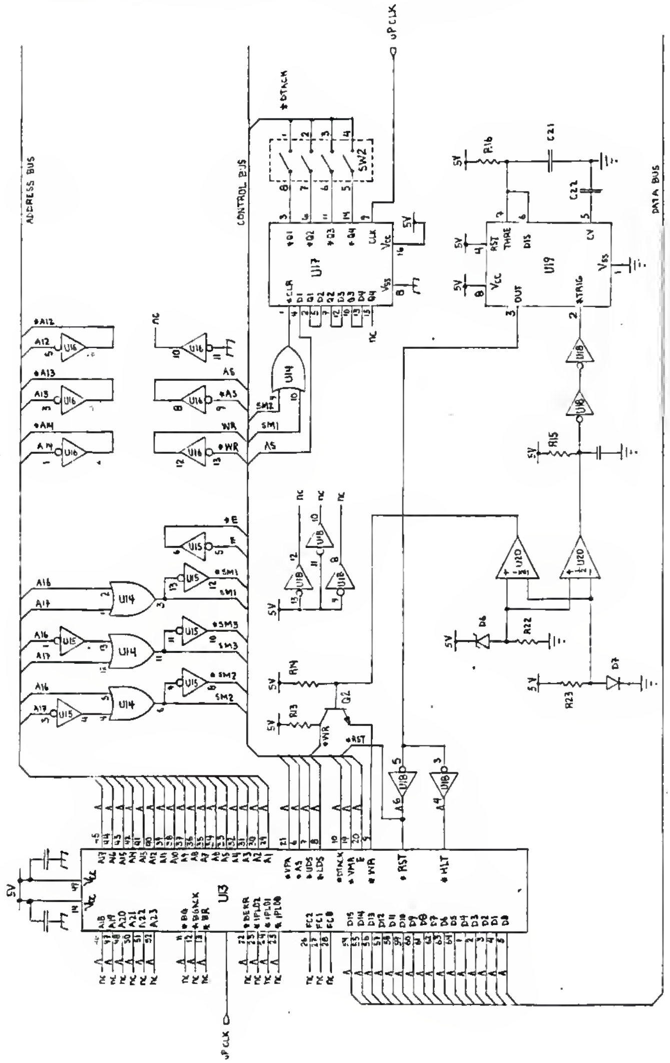
The 68000 microprocessor forms the heart of the harmonic analyzer's digital section. It provides the basis for almost all the control signals of the digital section. In addition it provides the control and signal processing capabilities needed to meld the rest of the components together. Figure 2.4 shows the microprocessor and its related circuitry.

The microprocessor is clocked at 4.9152 MHz. Its I/O lines are connected to the 5V supply through a set of 4.7K pullup resistors. This allows the processor to reliably interface with the CMOS circuitry that forms the majority of the analyzer's digital section.

Almost all of the control signals in the digital section of the analyzer are derived from the microprocessor. Those provided directly by the processor include *AS, *UDS, *LDS, *VMA, E and R/W. Signals provided indirectly include SM1, SM2, SM3, *VPA and *DTACK.

The control signals SM1, SM2 and SM3 are decoded from the address lines A17 and A16 by U14 and U15. These signals and their complements decode the system memory into 64KB regions. SM1 is used to enable ROM. SM2 is used to enable the I/O devices and is fed back to the *VPA input of the microprocessor to signal that this region uses a synchronous bus protocol instead of the asynchronous bus used by the rest of this system. This enables lower speed devices to be

Figure 2.4 : Microprocessor and Support Circuitry



interfaced to the 68000. SM3 is used to enable RAM.

The signal *DTACK is derived from *AS, SM1 and SM3 by U14 and U17. This signal marks the end of a bus cycle for the asynchronous bus. It is only used for the regions of memory reserved for RAM and ROM. The time for a data transfer can be set by switch, SW2. This allows the timing to be adjusted for various memory chips.

The reset signal, *RST, is not derived from any microprocessor signal. It is independently asserted on power up or whenever the supply voltage drops below 4.7V. This prevents the microprocessor from being put in a nonfunctional state by a power gliche. The comparator U20 performs the voltage sensing and the 555 timer and inverter chips, U19 and U18, provide the reset pulse of an appropriate length.

System Memory, EPROM:

The harmonic analyzer possesses 16K bytes of memory space devoted to ROM. The space is currently occupied by two 4Kx8 EPROMs, U21 and U22 as shown in Figure 2.5. These can be replaced by two 8K EPROMs if more program storage space is required.

The EPROMs block of memory is decoded by U23, a quad OR gate. The OR gates select the EPROM when *SM1 and *AS are active. The output of the individual EPROMs are enabled

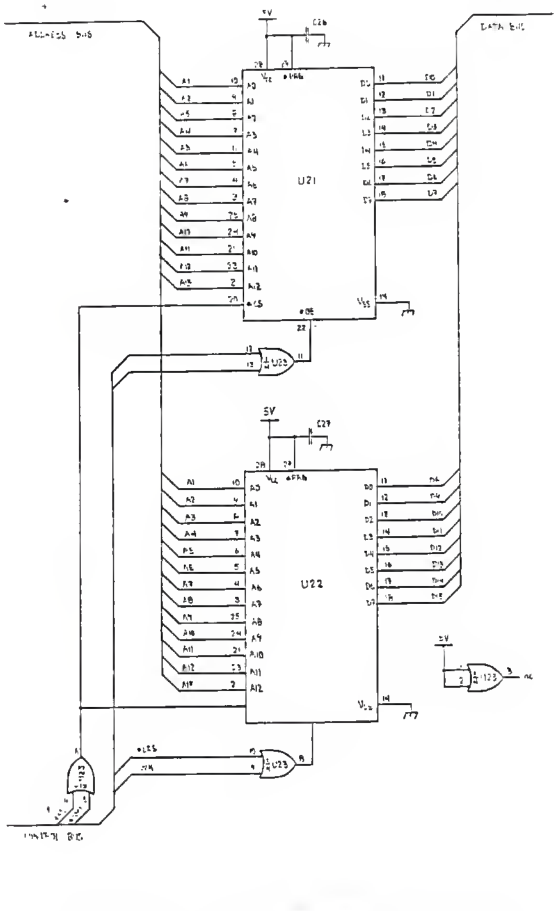


Figure 2.5 : System Memory, EPROM

when the data strobe to a chip is active and R/W is in the read mode. This prevents the EPROMs from placing data on the data bus and causing a bus collision if the EPROMs are inadvertently written to.

System Memory, RAM:

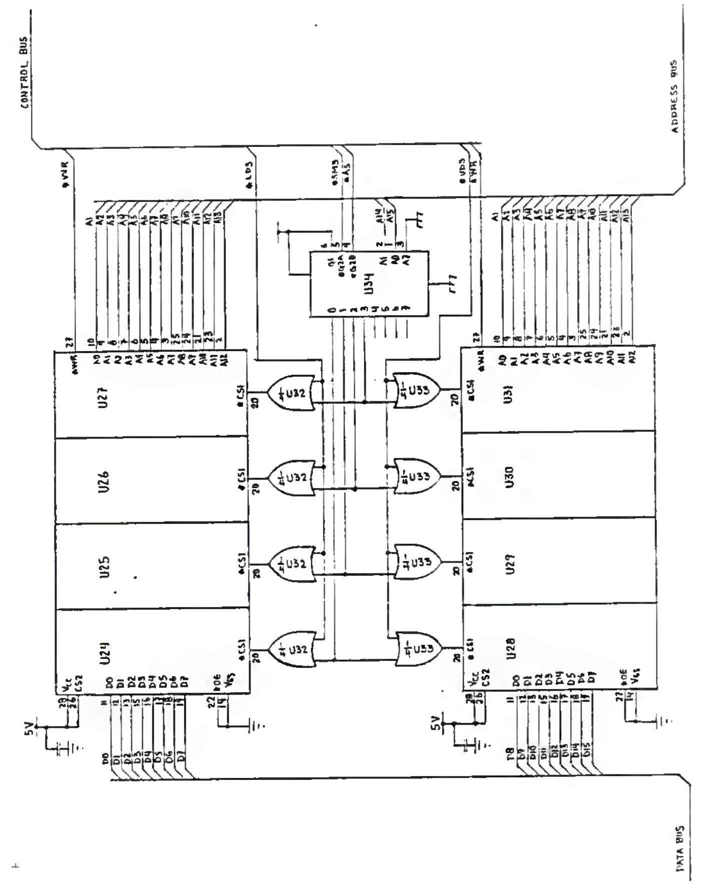
The harmonic analyzer possesses 64K bytes of static random access memory (RAM). It is structured as two 32K words due to the system's sixteen bit data bus. The RAM is battery backed, and so is nonvolatile as far as system power is concerned. Figure 2.6 shows the system RAM and associated memory decoding circuitry. Timing diagrams are shown in Appendix A.

The 64K bytes of RAM are made up of eight 6264 8Kx8 static RAM chips, U24 to U31. The memory block occupied by the RAM is $02xxxx.

The decoding is done using the signals *SM3 and *AS to enable an eight state decoder, U34, which decodes A14 and A15 using its low four states. This gives a control line for each 16K bytes in the $02xxxx block of memory addresses. These signals are then OR'ed with the data strobes, *UDS and *LDS, by U32 and U33 to get a chip enable for every 8K block of memory.

The grounds for the RAM chips are tied to the auxilary

Figure 2.6 : System Memory, RAM



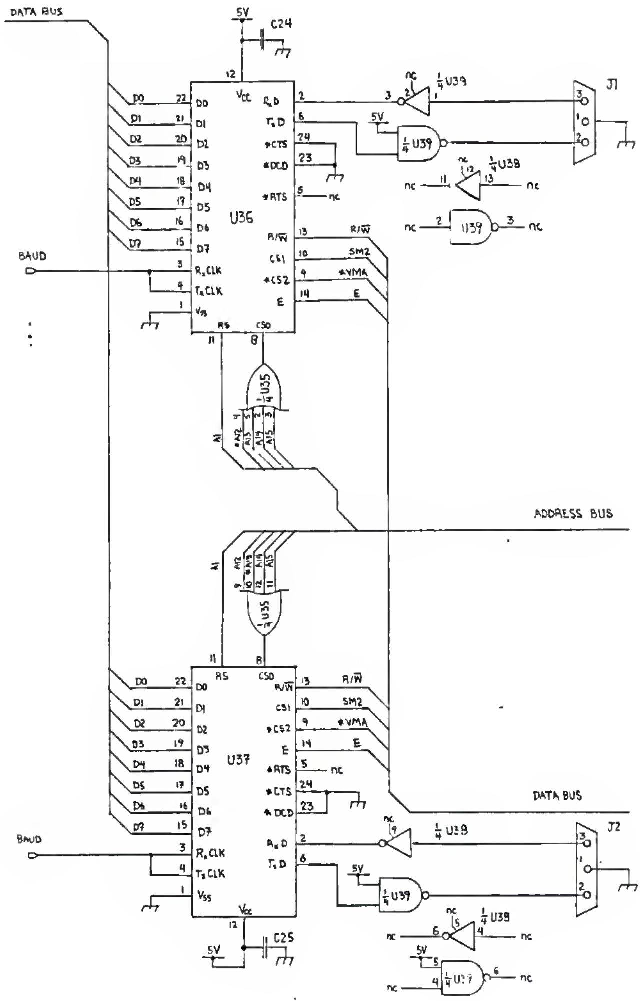
ground. This ground is only tied to the system ground when power is applied to the circuit. When power is down only devices on this ground possess a path for current flow and so can be backed up with a battery.

Serial Ports:

The harmonic analyzer possesses two channels of serial communications. The schematics for the serial ports are shown in Figure 2.7. This is the only method of communicating with the external world on the harmonic analyzer. They run under an asynchronous protocol at variable baud rates. The output signals are at standard RS-232 levels. RS-232 control signals such as Clear To Send (CTS) and Request To Send (RTS) are not used. Provisions have been made on the board so that these signals may be used if desired for at least one channel. This would allow the analyzer to control a modem.

The serial ports, U36 and U37, are 6850 ACIAs. The ACIAs have their memory space decoded by U35, a dual 4 input OR gate. The outputs of the OR gates are connected to one of the three chip selects of the ACIAs. The remaining two chip selects of the ACIAs are connected to *VMA and SM2. These are connected so that when all three signals are active an ACIA is enabled.

Figure 2.7 : Serial Ports and Support Circuitry



The RS-232 level shifting is done by U38 and U39, a 1489 quad receiver and a 1488 quad transmitter respectively. These have been set up so that the transmit data and receive data lines for each ACIA are converted to RS-232 levels. The other control signals are not presently used. Provisions have been made so that the control signals for one channel can be set up.

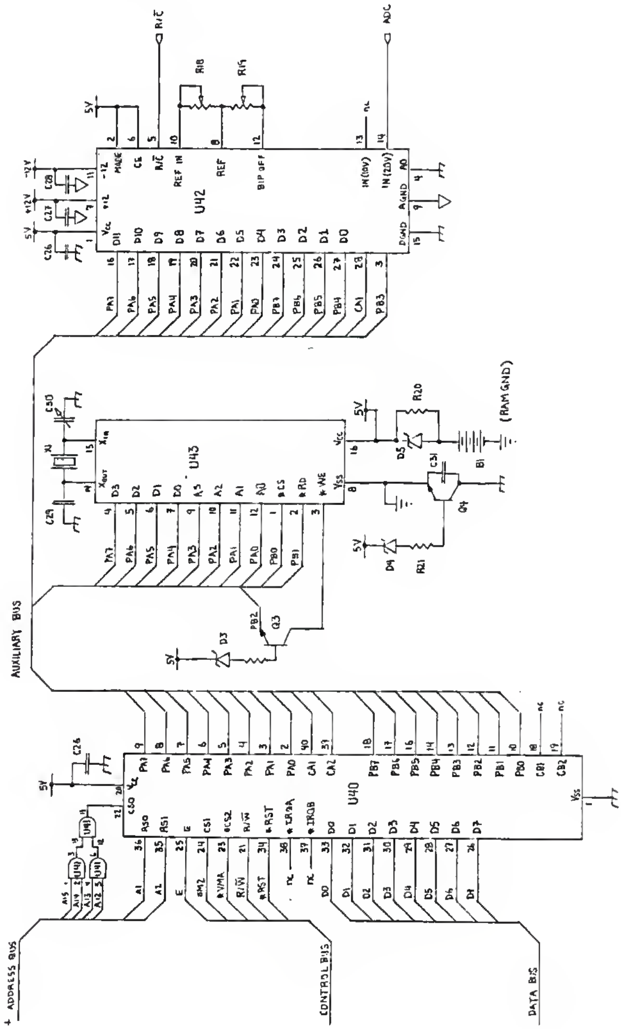
PIA,Real Time Clock,ADC

The peripherial interface adapter (PIA), U40, is the port through which communication with the real time clock (RTC) and the analog to digital converter (ADC) is done. Its bidirectional ports are used to form a software controlled auxiliary bus. This is shown in Figure 2.8. Table 2.1 shows the pin designations of the PIA.

The ADC, U42 and RTC, U43 were connected to the system in this way for a number of reasons. The RTC was attached to the system in this manner because its bus timing is too slow to operate on the actual microprocessor bus. The ADC is configured this way so that it can sample data at a frequency determined by an independent crystal clock and not interfere with bus activity.

The auxiliary bus is controlled via software to produce the signals to enable, address and read or write to the RTC

Figure 2.8 : PIA, Real Time Clock and ADC



or ADC. The only notable feature about the interface is the line controlling the *WR of the RTC. When power goes down the WR* of the RTC is forced inactive due to Q3 and an internal pullup on the *WR pin of the RTC. This prevents false writes to the clock when system power is going down.

The RTC is also battery powered so it will continue to function when system power is cutoff. It is disconnected from the system ground by Q3 when power fails. This keeps the backup battery from powering the rest of the system. The system RAM is also tied to this auxiliary ground so that it is provided with power when power fails.

TABLE 2.1 : PIA PIN DEFINITIONS

Control Pins:

CA1 (input)
conversion status from the A/D. A negative transition signals the end of conversion.

CA2 (output)
mux address control. low selects the voltage sensor, high selects the current sensor.

CB1 (unused)

CB2 (unused)

Port B Pins:

```
b7 (i) : a/d data, d3
b6 (i) : a/d data, d2
b5 (i) : a/d data, d1
b4 (i) : a/d data, d0

b3 (o) : a/d chip select, active low
b2 (o) : clock write strobe, active low
b1 (o) : clock read strobe, active low
b0 (o) : clock select, active low
```

Port A Pins:

```
a7 (i/o) : A/D data, d11; clock data, d3
a6 (i/o) : A/D data, d10; clock data, d2
a5 (i/o) : A/D data, d9 ; clock data, d1
a4 (i/o) : A/D data, d8 ; clock data, d0

a3 (i/o) : A/D data, d7 ; clock address, a3
a2 (i/o) : A/D data, d6 ; clock address, a2
a1 (i/o) : A/D data, d5 ; clock address, a1
a0 (i/o) : A/D data, d4 ; clock address, a0
```

PIA Register Locations:

```
01 f001  - port a data/direction register
01 f003  - port a control register
01 f005  - port b data/direction register
01 f007  - port b control register
```

CHAPTER 3 : HARMONIC ANALYZER SOFTWARE

Introduction

As with all microprocessor based pieces of equipment the software for the harmonic analyzer gives function to the form provided by the hardware. It provides the control for the sampling port, the signal processing capabilitites and the communication protocol necessary to relay this information.

The control program for the harmonic analyzer, ARDVARK, is written in 68000 assembly code. It was assembled using the Avocet MAC68K macroassembler. This assembler possesses many features that simplified software development. The most important of these was the ability to create relocatable linkable object modules. These could then be linked to produce the final object code. This allowed the development of ARDVARK in its present modular structure.

The structure of ARDVARK follows the form recommended for software development. The main program, ARDVARK, is a short piece of code that makes use of numerous subroutines to accomplish its tasks. These subroutines in turn call other subroutines which further divide the task until it has been reduced to simplest terms. Development of each module in this fashion made debugging of the program an easier task.

The subroutines tend to follow a general form. The register used for passing most arguments to and from a subroutine is d7 whenever possible. This was done in some cases where passing the argument in d7 was not the optimal solution in order to create a regular subroutine interface. Most of the routines save the contents of the microprocessor's register on the stack on entry. Any registers whose contents are changed by the routine are returned to their original state in exiting the routine. This makes the routines easy to utilize and prevents the occurrence of side effects. This in effect makes the I/O registers global variables and all other registers local variables in a routine.

Some of the routines that make up the harmonic analyzer's software package are not actual subroutines. They are instead program fragments. They return execution to a specific point rather than back to the routine that called them. The descriptions of these routines make note of this characteristic.

The remainder of this chapter discusses the various functions of ARDVARK. Table 3.1 shows a list of the various routines and a basic description of their functions. The routines are then discussed in more detail in the remainder of this chapter. To aid in this discussion the 68000 micro-processor's architecture is shown in Figure 3.1. For further information on the 68000 see [1]. This discussion deals with the routines on a flow chart level. The theory needed for understanding the functioning of a routine is included in the description of the routine when necessary. For even greater detail on the routines consult the actual source code listings in Appendix B.

```
     31              16 15       8 7         0
     ------------------------------------------
D0  |                |         |            |
D1  |                |         |            |
D2  |                |         |            |    Eight
D3  |                |         |            |    Data
D4  |                |         |            |    Registers
D5  |                |         |            |
D6  |                |         |            |
D7  |                |         |            |
     ------------------------------------------

     31              16 15       8 7         0
     ------------------------------------------
A0  |                |         |            |
A1  |                |         |            |
A2  |                |         |            |    Seven
A3  |                |         |            |    Address
A4  |                |         |            |    Registers
A5  |                |         |            |
A6  |                |         |            |
     ------------------------------------------

     31                                      0
     ------------------------------------------
    |          User stack pointer           |    Two
A7   ------------------------------------------  Stack
    |       Supervisor stack pointer        |    Pointers
     ------------------------------------------

     31                                      0
     ------------------------------------------
PC  |                                       |    Program
     ------------------------------------------  Counter

                     15       8 7            0
                     ----------------------
SR                   | System|   User     |      Status
                     ----------------------      Register
```

Figure 3.1 : The Programming Model For The 68000

TABLE 3.1 : ARDVARK routines and functions

ARDVARK:	main program.
ASCII.HEX:	convert ASCII coded hex digits to hexadecimal form.
CLOCK.READ:	reads a register of the real time clock.
CLOCK.WRITE:	writes to a register of the real time clock
DISP.REG:	outputs the contents of a uP register in ASCII coded hexadecimal digits through the secondary serial port.
DISP.STRING:	outputs characters in memory through the secondary serial port until an ASCII null is reached.
DOWNLOAD:	routine for downloading S-record files from a host computer.
FFT:	performs an FFT on integer data stored in memory.
GETCHAR:	gets a character from the secondary serial port.
GET.DATA:	gets one cycle of current and voltage data and stores it in memory
GET.TIME:	reads the time from the real time clock.
GO:	begins execution at a specified memory location.
HEX.ASCII:	converts a hex nybble to its equivalent ASCII code.
HEX.CHECK:	checks to see if a ASCII character corresponds to a hex nybble.

TABLE 3.1 (continued)

IN:	gets a character from the primary serial port and echoes it back.
INITIALIZE:	sets the time of the real time clock to a specified time and initializes system variables.
MEMORY:	allows the examination and alteration of memory contents.
OUT:	outputs a character through the primary serial port and waits for the character to be echoed back.
PUTCHAR:	outputs a character through the second serial port.
QK.FFT:	gets data, performs FFT and outputs results through the primary serial port.
SAMPLE:	controls the process of sampling and storing data.
SEND.DATA:	sends the stored data in the harmonic analyzers memory out the primary serial port.
SEND.TIME:	sends a four character string that gives the time read from harmonic analyzer's real time clock.
TRACE:	single steps through a program while allowing examination of all microprocessor registers.

ARDVARK:

ARDVARK is the main routine for the harmonic analyzer.

It performs all the control, computation and communication functions of the harmonic analyzer. It performs these functions by making calls to lower level routines. For this reason ARDVARK is basically a shell that handles the routing of program flow.

Figure 3.2 shows a flowchart of ARDVARK. On system bootup execution begins at the beginning of ARDVARK. ARDVARK first resets and configures the serial ports (6850 ACIA's) and then configures the parallel port (6821 PIA).

The cyclic portion of ARDVARK begins at this point. The status of the first serial port is checked to see if a byte has been received. If so the port is read and the character is decoded to find which function the harmonic analyzer should perform. Program flow is then routed to the appropriate subroutine. If the character received corresponds to no function then execution proceeds as if no code were received.

If no character were received ARDVARK then reads the time from the real time clock. If the clock setting is midnight (00:00) then the variable sys.go is made true. Sys.go is a variable that tells the system that sampling is enabled. Sampling is enabled at midnight this way. ARDVARK then checks the clock to see if the time is on the hour. If it is then ARDVARK waits until the next minute to take a

Figure 3.2: Flowchart of the control program ARDVARK.

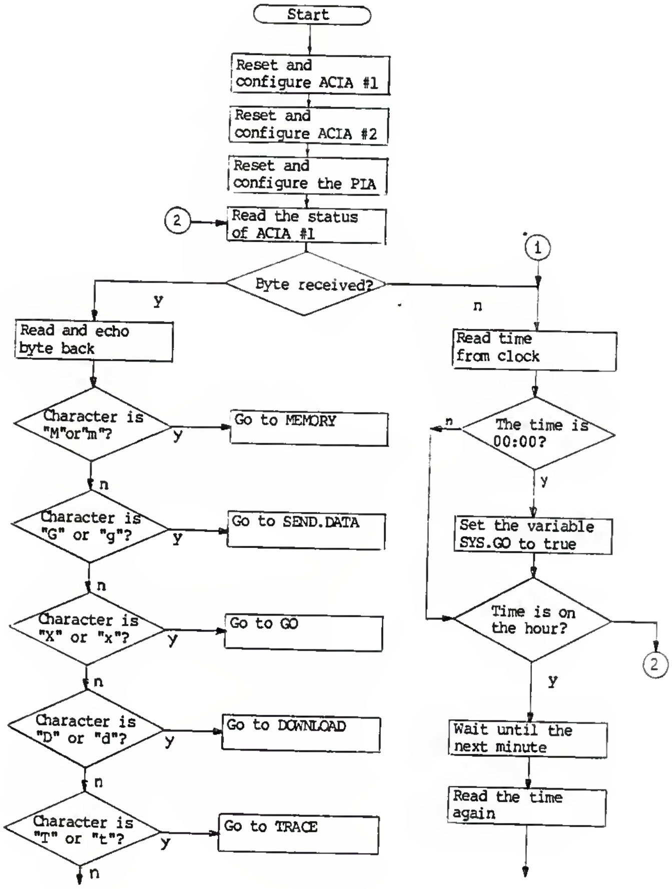
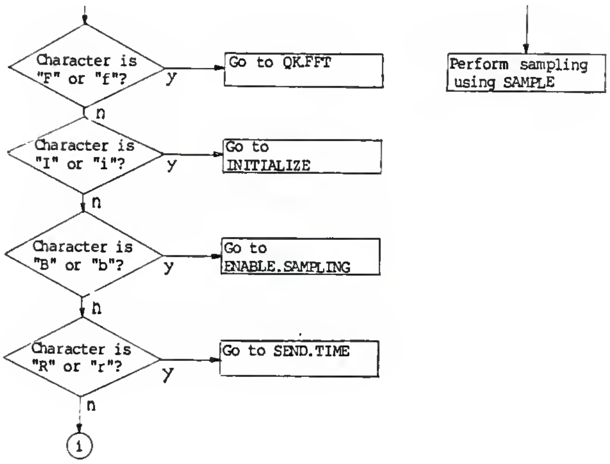


Figure 3.2: Flowchart of the control program ARDVARK.



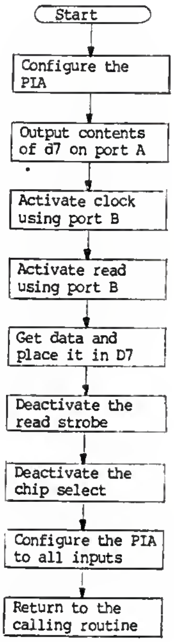
sample. ARDVARK is unavailable for external communication for this minute. This technique prevents overruns in the data sampling. If the time is not on the hour ARDVARK then goes back to the point where it reads the status of the serial port.

CLOCK.READ:

CLOCK.READ is a primitive that is used to read the data in a clock register. This routine can be used repetitively to read the time from the clock. It provides the control for the auxiliary bus needed to read from the real time clock. The number of the clock register to be read is passed in register d7 of the microprocessor. The data read is returned in d7 also. No other register contents are altered. Figure 3.3 flowcharts the operation of CLOCK.READ.

CLOCK.READ first configures the parallel port to set up the appropriate input and output lines. The register number is then output through the parallel port so that it is on the clock's address lines. CLOCK.READ then runs the control strobes, which are generated by the parallel port, to get the data from the clock. Since the data is read into bits 4-7 CLOCK.READ shifts it down to put it in bits 0-3. CLOCK.READ then configures the parallel port as it was on entry and returns to the calling routine.

Figure 3.3: Flowchart for the routine CLOCK.READ


Start
Configure the PIA
Output contents of d7 on port A
Activate clock using port B
Activate read using port B
Get data and place it in D7
Deactivate the read strobe
Deactivate the chip select
Configure the PIA to all inputs
Return to the calling routine

CLOCK.WRITE:

This routine is another primitive. It is used to write data to a register of the real time clock. It can be used repetitively to set the clock to a given time. It works in much the same way as CLOCK.READ. A flowchart of its operation is shown in Figure 3.4.

The PIA is configured so that the A port is all outputs. The register number is passed in d7 output so that it is on the address lines of the clock. The data is output so that it is on the clock's data lines. The control strobes for the clock are then toggled so that the data is latched into the clock. The A port of the PIA is then configured as all inputs again. Execution then returns to the calling routine.

DISP.REG:

DISP.REG outputs the contents of a register through the second serial port. The contents of the register to display are placed in d7 for the call to DISP.REG. The register's contents are converted to ASCII coded hexadecimal and output though the secondary serial port. The data can be displayed on a terminal. No register contents are altered by this routine. A flowchart of DISP.REG is shown in Figure 3.5.

Figure 3.4: Flowchart for the routine CLOCK.WRITE

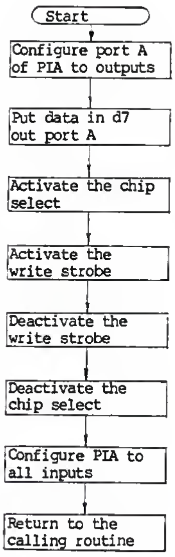
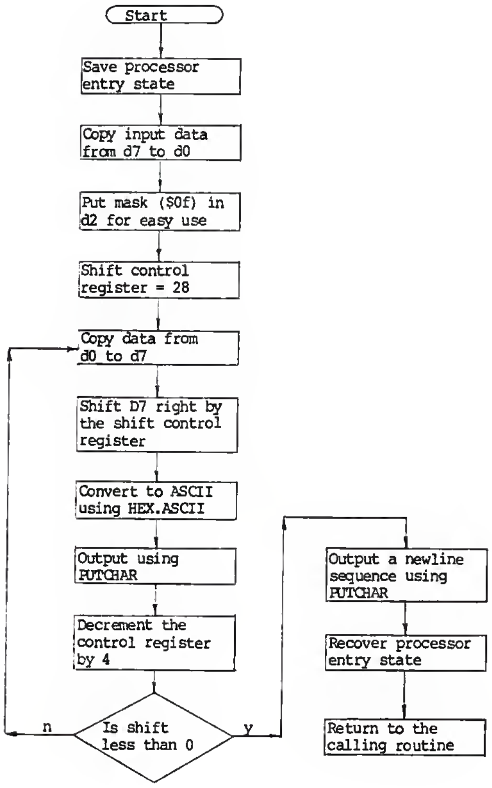

Start
Configure port A of PIA to outputs
Put data in d7 out port A
Activate the chip select
Activate the write strobe
Deactivate the write strobe
Deactivate the chip select
Configure PIA to all inputs
Return to the calling routine

Figure 3.5: Flowchart for the routine DISP.REG

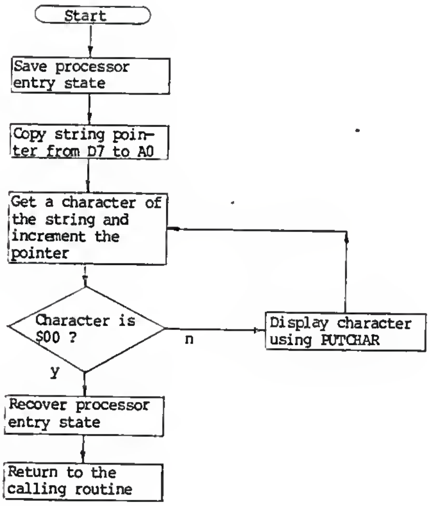

Start
Save processor entry state
Copy input data from d7 to d0
Put mask ($0f) in d2 for easy use
Shift control register = 28
Copy data from d0 to d7
Shift D7 right by the shift control register
Convert to ASCII using HEX.ASCII
Output using PUTCHAR
Decrement the control register by 4
Is shift less than 0
n
y
Output a newline sequence using PUTCHAR
Recover processor entry state
Return to the calling routine

The contents of the microprocessor's registers are saved on entry to DISP.REG. The input data is saved in another register. Each successive nybble from the most to least significant is treated in the following manner. The input data is copied from the storage register back into d7. The nybble is shifted to the least significant position in d7. The nybble is then converted to ASCII by calling the routine HEX.ASCII. The resulting ASCII character is then sent out to the second serial port using the routine PUTCHAR. The shift control register is then adjusted so the next most significant nybble is shifted to the bottom of the register. When all of the input data has been converted and sent a carriage return and a linefeed are sent out using PUTCHAR. The processor's entry state is then restored and DISP.REG returns to the calling routine.

DISP.STRING:

DISP.STRING is a routine to send the contents of memory from a starting memory location, passed in d7, to an ending point signaled by an ASCII null, $00. The data is sent out the second serial port. The data sent in this fashion is generally textual, hence the subroutines name. No register contents are altered by this routine. Figure 3.6 shows a flow chart of DISP.STRING.

Figure 3.6: Flowchart for the routine DISP.STRING


Start
Save processor entry state
Copy string pointer from D7 to A0
Get a character of the string and increment the pointer
Character is $00 ?
n
Display character using PUTCHAR
y
Recover processor entry state
Return to the calling routine

The entry state of the microprocessor's registers is saved. The starting location is then copied into an address register. A byte is read from memory into a data register and the address register is incremented to point at the next byte of memory. If the byte is not an ASCII null ($00) it is sent out the second serial port using PUTCHAR. The process continues until a null is encountered. If an ASCII null is encountered then processor's entry state is restored and program flow returns to the calling routine.

DOWNLOAD:

DOWNLOAD allows S-record files to be downloaded from a host computer to the harmonic analyzer via a serial communications link. The S-record file format is the format used to store object code produced as an assemblers output. This format is the one specified by Motorola. For further information see [2]. The data is stored in hexidecimal form at the locations specified for it in the file itself. Using this routine programs can be downloaded to the harmonic analyzer and tested. Figure 3.7 shows a flow chart of DOWNLOAD.

DOWNLOAD operates in the following manner. It first searches for an S in the incoming data stream. When an S is found the next character is checked to see if it is a 1,2 or

Figure 3.7: Flowchart for the routine DOWNLOAD

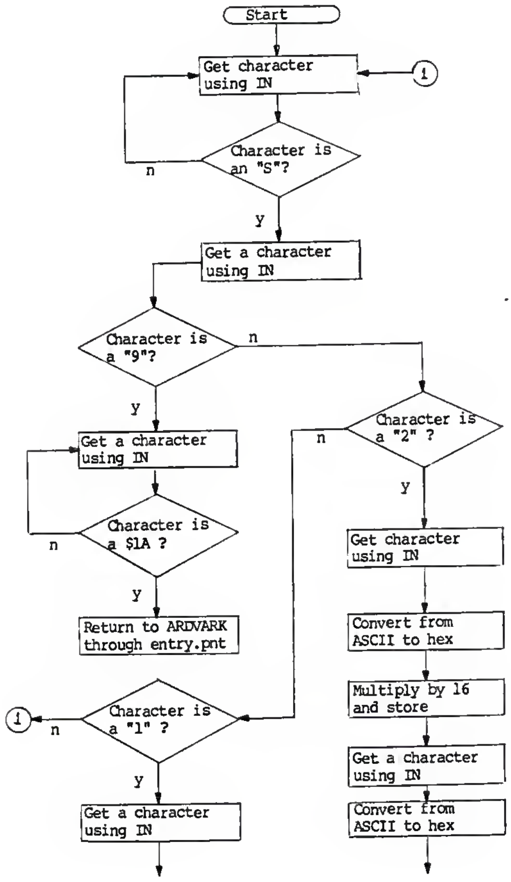


Figure 3.7: Flowchart for the routine DOWNLOAD

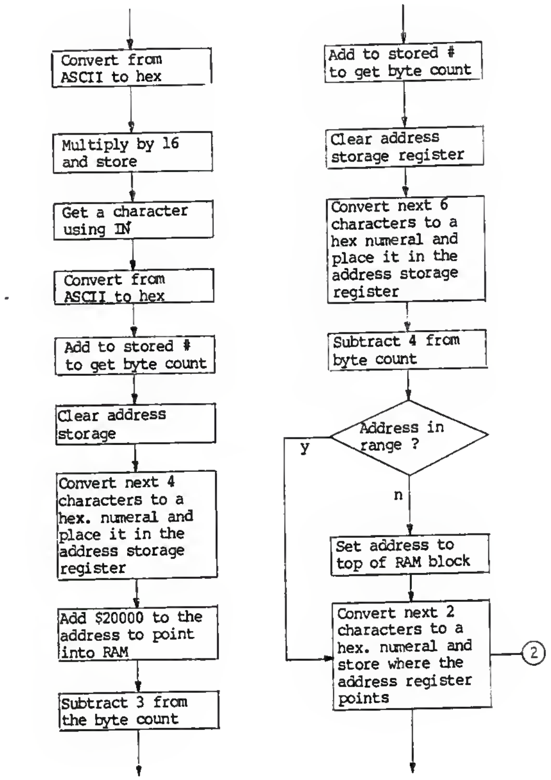
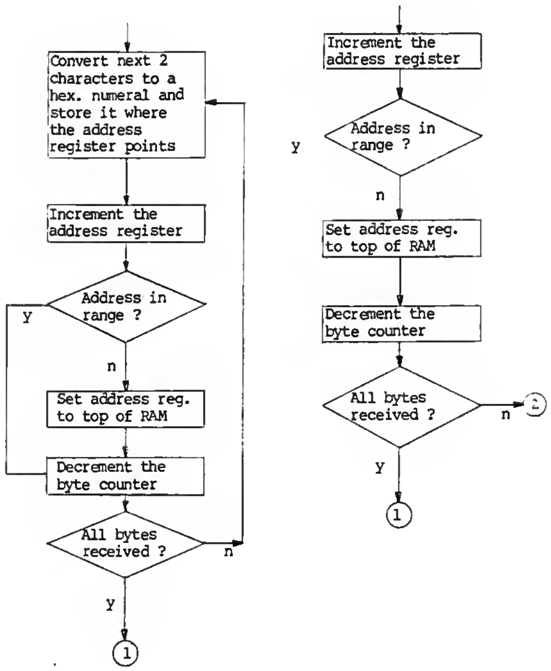

Convert from ASCII to hex
Multiply by 16 and store
Get a character using IN
Convert from ASCII to hex
Add to stored # to get byte count
Clear address storage
Convert next 4 characters to a hex. numeral and place it in the address storage register
Add $20000 to the address to point into RAM
Subtract 3 from the byte count
Add to stored # to get byte count
Clear address storage register
Convert next 6 characters to a hex numeral and place it in the address storage register
Subtract 4 from byte count
Address in range ?
y
n
Set address to top of RAM block
Convert next 2 characters to a hex. numeral and store where the address register points
2

Figure 3.7: Flowchart for the routine DOWNLOAD


Convert next 2 characters to a hex. numeral and store it where the address register points
Increment the address register
Address in range ?
y
n
Set address reg. to top of RAM
Decrement the byte counter
All bytes received ?
n
y
1
Increment the address register
Address in range ?
y
n
Set address reg. to top of RAM
Decrement the byte counter
All bytes received ?
n
2
y
1

9 and program flow is routed to the S1,S2 or S9 line decoding portion of the routine, respectively. If the character after the S is not one of these three then download resumes looking for an S again in the same manner.

The S1 line decoding section functions as follows. The first two ASCII characters are read in and converted to a hexidecimal numeral with the first character being considered the most significant digit. This gives the number of character pairs or data bytes remaining in the line. The next four characters received are the ASCII coded address. These are similarly converted into a hexidecimal number and stored as the data pointer. The data byte counter is then reduced by three, two for the address and one for the checksum at the end of the line. The next pair of characters is converted to a hexidecimal numeral and stored at the position pointed at by the data pointer. The data pointer is then incremented and checked to see if it points at physical memory. If not, the data pointer is loaded with the last byte in RAM. The data counter is decremented and if there are more characters left in the line the process continues. When the line is completed program flow returns to the start of the routine.

The S2 line data line decoding routine functions as follows. As in the S1 decoding routine the first two characters are converted to a hexidecimal numeral that is the number of data bytes represented by the rest of the line. This is stored as the byte counter. The next six characters are converted from an ASCII coded hexidecimal number to binary. This is the address at which the data placement is to start. This is saved as the data pointer. If the address is out of range of the harmonic analyzers memory then the address of the last byte of RAM is placed in the data pointer. This allows data to be placed anywhere in the 68000's addressing range. Data is converted and stored in the same way as in the S1 decoding routine. When the end of the line is reached program flow returns to the beginning of the routine.

The S9 line is the terminating line of an S-record file. The routine for decoding it merely waits until all the data has been sent and only filler data from a disk file is being sent. This is sent as $1A's on the host computer used to develop the harmonic analyzer. When this state is reached program flow returns to ARDVARK through the node entry.pnt.

FFT:

The routine FFT is the heart of the harmonic analyzer's software package. It performs a decimation in frequency fast Fourier transform (FFT) on a complex input sequence. The input data is assumed to be in offset integer. A pointer to the memory block storing the samples is passed in register a6 of the processor. No registers are altered by this routine.

The fast Fourier transform (FFT) is actually a discrete Fourier transform (DFT) calculated by a fast algorithm. The DFT is calculated by equation 1.

$$X(n) = \sum_{m=0}^{N-1} x(m) e^{-j2\pi nm/N} \quad \ldots(3.1)$$

where:

$n = 0,1,\ldots,N-1$

$m = 0,1,\ldots,N-1$

N = number of points

$X(n)$ = sequence of Fourier coefficients

$x(n)$ = input sequence

The DFT shows the distribution of the energy of a signal over a set of discrete frequencies. For further information on the properties of the DFT see references [3], [4] and [5]. A fast algorithm can be used to compute the DFT with a large time savings over the straight forward computation of the equation above. This computation is called the fast Fourier transform or FFT.

The fast algorithm used to compute the DFT on the harmonic analyzer is the Cooley-Tukey decimation in frequency FFT. The decimation in frequency algorithm functions only for sequences of length 2 . If a sequence is not of length 2 then it must be zero padded or truncated so that it is.

The decimation in frequency algorithm can be derived from the DFT in the following manner. For a more indepth treatment see [3], [4] and [5]. The equation for calculating the DFT can be rewritten by dividing the input sequence in two.

$$X(n) = \sum_{m=0}^{N/2-1} x(m)e^{-j2\pi nm/N} + \sum_{m=N/2}^{N-1} x(m)e^{-j2\pi nm/N} \qquad \ldots(3.2)$$

$$X(n) = \sum_{m=0}^{N/2-1} x(m)e^{-j2\pi nm/N} +$$

$$e^{-j(2\pi/N)(N/2)n} \sum_{m=0}^{N/2-1} x(m+N/2)e^{-j2\pi nm/N}$$

But $e^{-j(2\pi/N)(N/2)n} = e^{-j\pi n} = (-1)^n$

$$X(n) = \sum_{m=0}^{N/2-1} [x(m) + (-1)^n x(m+N/2)]\, e^{-j2\pi nm/N}$$

This can be rewritten for the odd and even DFT coefficients.

For the even coefficients this becomes:

$$X(2c) = \sum_{m=0}^{N/2-1} [x(m)+x(m+N/2)]e^{-j(2\pi/N)2cn}$$

$$X(2c) = \sum_{m=0}^{N/2-1} [x(m)+x(m+N/2)]e^{-j2\pi nc/(N/2)}$$

...(3.2a)

For the odd coefficients this becomes:

$$X(2c+1) = \sum_{m=0}^{N/2-1} [x(m)-x(m+N/2)]e^{-j(2\pi/N)(2c+1)n}$$

$$X(2c+1) = \sum_{m=0}^{N/2-1} [x(m)-x(m+N/2)]e^{-j2\pi m/(N/2)} e^{-j2\pi nc/(N/2)}$$

where $c = 0,1,\ldots,N/2-1$

...(3.2b)

The two sequences can be seen to be DFT's themselves. The even DFT coefficients can be seen to be the result of taking the DFT of the N/2 point sequence $\{x(m)+x(m+N/2)\}$ where $m=0,\ldots,N/2-1$. The odd DFT coefficients are the result of taking the DFT of the N/2 point sequence{ [x(m)-x(m+N/2)]exp(-j(2 m)/(N/2) } where m=0,...,N/2-1. These DFT's can be broken up in the same manner. This continues as many times as necessary to reduce the sequences to one point sequences. The one point sequences are the DFT coefficients, X(n).

The problem with this technique is that the results are not in the correct order. Consider an eight point DFT done via this technique. The results of splitting the sequence yields a sequence whose DFT is X(0), X(2), X(4), X(6) and another sequence whose DFT is X(1), X(3), X(5), X(7) of the original sequence. This is shown in Figure 3.8. The figure used to explain this is a signal flow graph. An explanation of signal flow graphs is given in Appendix D.

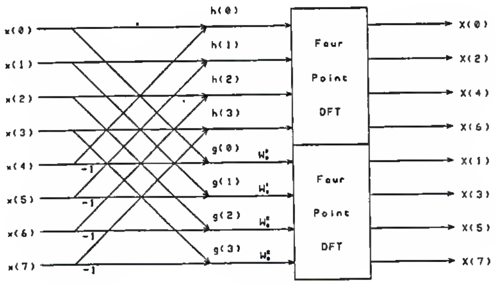


Figure 3.8: Eight point DFT divided into two four point DFT's.

Performing the division of the sequences again on the four point sequences further scrambles the results. This is shown in Figure 3.9 for the subsequence marked h(0), h(1), h(2) and h(3) above.

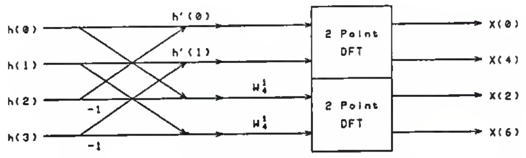


Figure 3.9 : A four point DFT divided into two two point DFT's.

The two point FFT does not further scramble the results. This is because the sequences have only one point so division into the odd and even sequences does not reorder the elements. Also the two point FFT is actually the DFT. A two point DFT is shown below in Figure 3.10.

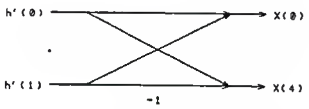


Figure 3.10 : A two point DFT

An unscrambling action is needed to place the resulting coefficients in the proper order. The sequence resulting from the FFT, R(n), are actually the scrambled results of the DFT, X(n). The relationships for an eight point FFT are shown in Figure 3.11. Due to the symmetrical relationships of the unscrambling action it is often referred to as the "butterfly transfers."

The output of the FFT can be placed in order by swapping some of the elements. Swapping R(1) with R(4) and R(3) with R(6) will put the output of an eight point FFT into the proper order. (See Figure 3.11.) The pairs of elements to be swapped can be determined by bit reversing the indices of the FFT coefficients.

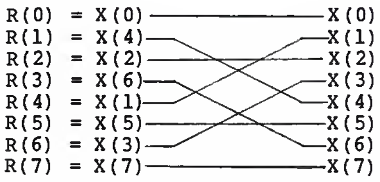


Figure 3.11 : The relationship between the output of the FFT, R(n), and the DFT, X(n) for an eight point sequence.

The bit reversal is performed in the following way. Express the index of a coefficient in binary form. The number of bits used is determined by the length of the sequence. For an eight point DFT 3 bits are needed to express all possible indices. For R(1) the index would be 001. Bit reversing this gives 100, or 4. R(1) would then be swapped with R(4) to unscramble the output. Some numbers, such as 2 and 5 in this case, have the same value when bit reversal is performed. This means that the element is already in its correct location in the sequence.

This process continues until all related pairs of elements are swapped. Once all the pairs are swapped the output of the FFT is in the correct order.

The signal flow graph showing the calculations needed for an eight point FFT are shown in Figure 3.12. Figure 3.13 shows a flow chart of the routine FFT used to calculate a 128 point FFT. For comparison the listing of a FORTRAN program that performs a decimation in frequency FFT is given in Listing 3.1.

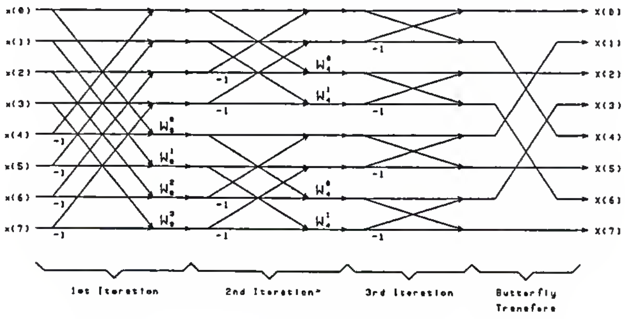


Figure 3.12: Signal flow graph of an 8 point DFT

Figure 3.13: Flowchart of the routine FFT

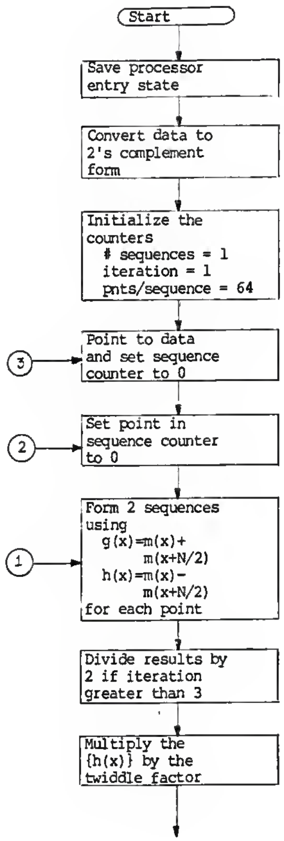

Start
Save processor entry state
Convert data to 2's complement form
Initialize the counters
# sequences = 1
iteration = 1
pnts/sequence = 64
Point to data and set sequence counter to 0
3
Set point in sequence counter to 0
2
Form 2 sequences using
g(x)=m(x)+ m(x+N/2)
h(x)=m(x)- m(x+N/2)
for each point
1
Divide results by 2 if iteration greater than 3
Multiply the {h(x)} by the twiddle factor

Figure 3.13: Flowchart of the routine FFT

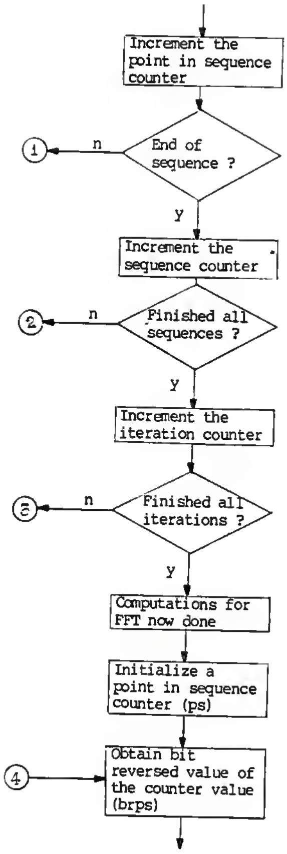
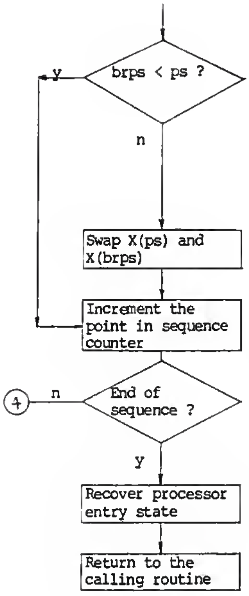

Increment the point in sequence counter
End of sequence ?
n
1
y
Increment the sequence counter
Finished all sequences ?
n
2
y
Increment the iteration counter
Finished all iterations ?
n
3
y
Computations for FFT now done
Initialize a point in sequence counter (ps)
Obtain bit reversed value of the counter value (brps)
4

Figure 3.13: Flowchart of the routine FFT


brps < ps ?
y
n
Swap X(ps) and X(brps)
Increment the point in sequence counter
End of sequence ?
4
n
y
Recover processor entry state
Return to the calling routine

Butterfly transfers

Listing 3.1: FFT routine in FORTRAN

```
C******************************************************************************
        FAST FOURIER TRANSFORM
C
C       DG FORTRAN 5 SOURCE FILENAME:            CFFT.FR
C
C       DEPARTMENT OF ELECTRICAL ENGINEERING     KANSAS STATE UNIVERSITY
C
C       REVISION        DATE                     PROGRAMMER
C       --------        ----                     ----------
C       00.0            FEB 02, 1980             FRED RATCLIFFE
C
C******************************************************************************
C
C       CALLING SEQUENCE
C
C               CALL FFT(X,N,INV)
C
C       PURPOSE
C
C               FAST FOURIER TRANSFORMATION
C
C       ROUTINE(S) CALLED BY THIS ROUTINE
C
C               NONE
C
C       ARGUMENT(S) REQUIRED FROM THE CALLING ROUTINE
C
C               X       -       COMPLEX VECTOR TO BE TRANSFORMED
C               N       -       NUMBER OF POINTS TO BE TRANSFORMED
C                               (MUST BE A POWER OF TWO)
C               INV     -       INV=0 -> FORWARD TRANSFORM
C                               INV=1 -> INVERSE TRANSFORM
C
C       ARGUMENT(S) SUPPLIED  TO  THE CALLING ROUTINE
C
C               X       -       COMPLEX TRANSFORMED VECTOR
C
C******************************************************************************
C
C       NOTE 1: This subroutine makes no checks on the validity
C               of the data supplied by the calling routine.
C
C       NOTE 2: Argument(s) supplied by the calling routine are
C               not modified by this subroutine.
C
C       NOTE 3: This subroutine was obtained from 'ORTHOGONAL TRANSFORMS FOR
C               DIGITAL SIGNAL PROCESSING' by N. Ahmed.
C
C******************************************************************************
C
        SUBROUTINE FFT(X,N,INV)
        COMPLEX X(1),W,T
        ITER=0
        IREM=N
 10     IREM=IREM/2
        IF (IREM.EQ.0) GO TO 20
```

Listing 3.1: FFT routine in FORTRAN

```
          ITER=ITER+1
          GO TO 10
[illegible] CONTINUE
          S=-1
          IF (INV.EQ.1) S=1
          NXP2=N
          DO 50 IT=1,ITER
          NXP=NXP2
          NXP2=NXP/2
          WPWR=3.141592/FLOAT(NXP2)
          DO 40 M=1,NXP2
          ARG=FLOAT(M-1)*WPWR
          W=CMPLX(COS(ARG),S*SIN(ARG))
          DO 40 MXP=NXP,N,NXP
          J1=MXP-NXP+M
          J2=J1+NXP2
          T=X(J1)-X(J2)
          X(J1)=X(J1)+X(J2)
   40     X(J2)=T*W
   50     CONTINUE
          N2=N/2
          N1=N-1
          J=1
          DO 65 I=1,N1
          IF(I.GE.J) GO TO 55
          T=X(J)
          X(J)=X(I)
          X(I)=T
   55     K=N2
   60     IF(K.GE.J) GO TO 65
          J=J-K
          K=K/2
          GO TO 60
   65     J=J+K
          IF (INV.EQ.1) GO TO 75
          DO 70 I=1,N
   70     X(I)=X(I)/FLOAT(N)
   75     CONTINUE
          RETURN
          END
```

GET.CHAR:

This routine gets a character from the second serial port. The character read in is returned in the low byte of d7. No other registers are altered by this routine. Figure 3.14 shows a flowchart of GET.CHAR.

The status of the second serial port is read into d7. If no character has been received the status is checked until one has been received. When a character has been received it is read into d7 and GET.CHAR returns to the calling routine.

GET.DATA:

This routine controls the data sampling process. It samples one cycle of voltage waveform and then one cycle of current waveform. The data resulting from this process is stored in two sequential blocks of memory set aside for this purpose. (See the system memory map, Appendix C.) Execution returns to the calling routine at the end of the sampling process. No registers are altered by this routine. Figure 3.15 shows the flow chart of GET.DATA's operations.

GET.DATA operates in the following manner. The registers whose contents are changed in this routine are saved on the stack for recovery later. The PIA is configured to read data from the analog to digital converter (ADC) . A pointer

Figure 3.14: Flowchart of the routine GETCHAR

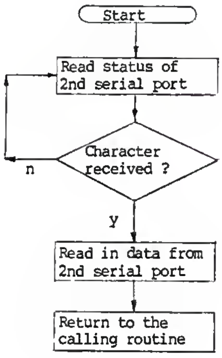

Start
Read status of 2nd serial port
Character received ?
n
y
Read in data from 2nd serial port
Return to the calling routine

Figure 3.15: Flowchart of the routine GET.DATA

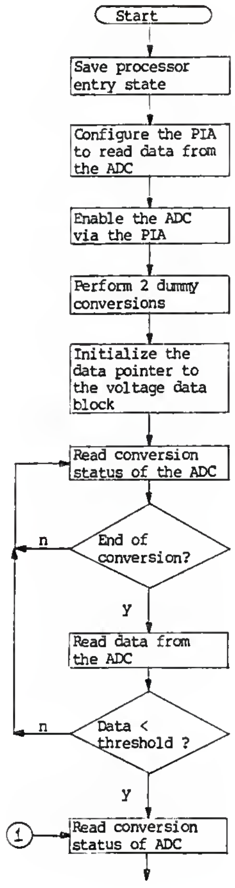

Start
Save processor entry state
Configure the PIA to read data from the ADC
Enable the ADC via the PIA
Perform 2 dummy conversions
Initialize the data pointer to the voltage data block
Read conversion status of the ADC
End of conversion?
n
y
Read data from the ADC
Data < threshold ?
n
y
1
Read conversion status of ADC

Figure 3.15: Flowchart of the routine GET.DATA

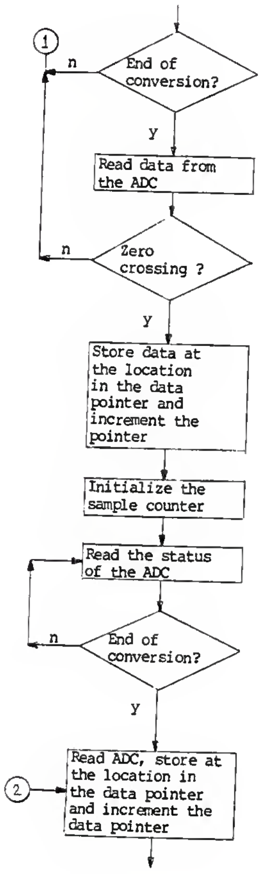

1
n
End of conversion?
y
Read data from the ADC
n
Zero crossing ?
y
Store data at the location in the data pointer and increment the pointer
Initialize the sample counter
Read the status of the ADC
n
End of conversion?
y
2
Read ADC, store at the location in the data pointer and increment the data pointer

Figure 3.15: Flowchart of the routine GET.DATA

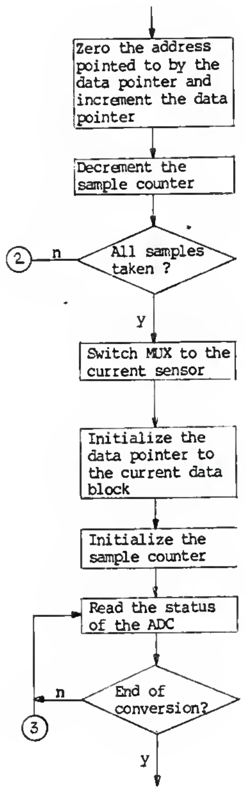
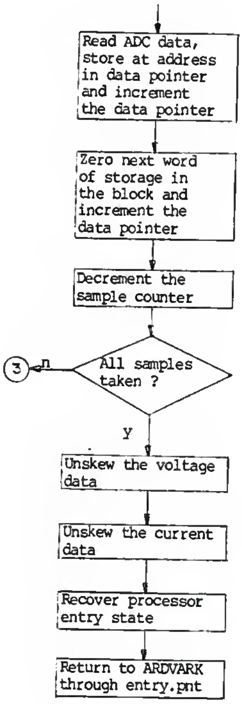

Zero the address pointed to by the data pointer and increment the data pointer
Decrement the sample counter
All samples taken ?
n
2
y
Switch MUX to the current sensor
Initialize the data pointer to the current data block
Initialize the sample counter
Read the status of the ADC
End of conversion?
n
3
y

Figure 3.15: Flowchart of the routine GET.DATA


Read ADC data, store at address in data pointer and increment the data pointer
Zero next word of storage in the block and increment the data pointer
Decrement the sample counter
All samples taken ?
n
3
y
Unskew the voltage data
Unskew the current data
Recover processor entry state
Return to ARDVARK through entry.pnt

is initialized to point to the voltage data storage block. The ADC is then enabled and two dummy conversions are performed.

GET.DATA then functions as a software zero crossing detector. Data storage does not begin until a zero crossing is detected. The zero crossing detector first waits until the voltage waveform has reached a negative threshold. This ensures that the detector will reliably monitor the zero crossing. It also means that sampling will not occur until an AC voltage of a large enough magnitude is monitored. The waveform is then monitored until a zero crossing is detected. One hundred and twenty eight samples are then obtained and stored in the block beginning where the data pointer pointed. The PIA is then used to switch the channel of the multiplexer. This connects the current sensor to the ADC. One hundred and twenty eight samples of current data are then taken. These are stored in a block of memory directly after the voltage data.

The output of the 12 bit ADC is stored in the high 12 bits of the 16 bit word, not the low twelve bits. For this reason the data is read back, shifted down four bits and all unessential bits cleared. This is done for both the voltage and current data.

Once the data is shifted GET.DATA prepares to return to the calling routine. The entry values of altered registers are recovered from the stack. Execution then returns to the calling routine.

GET.TIME:

This routine reads the time from the harmonic analyzers real time clock. It returns a word made up of four BCD digits that represent the hours and minutes. This word is returned to the calling routine in register d7. No registers other than d7 are altered by this routine. Figure 3.16 shows a flow chart of GET.TIME.

GET.TIME operates as follows. The registers changed in the routine are saved on the stack. The units of minutes register is read using the routine READ.CLOCK. The value is stored for later use. A register is then cleared to store the time. The tens of hours register is read from the clock and added to the time storage register. Since the time data is in the low nybble of the I/O register, d7, this places the time data in the low nybble of the time storage register. The time storage register is then shifted left by four bits. This clears off the low nybble of the time storage register so that another nybble can be added into that location. The same procedure is used to read in the data from the units of hours and tens of minutes registers of the clock. The units of minutes information is treated the same way except that the time storage register is not shifted once the data is added in. The value read in from the units of minutes register is then compared with its initial value

Figure 3.16: Flowchart of the routine GET.TIME

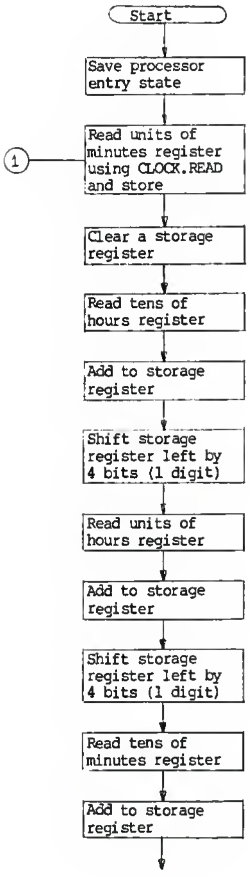
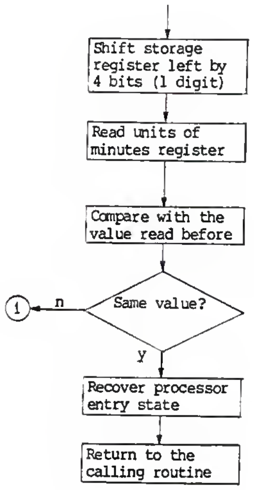

Start
Save processor entry state
Read units of minutes register using CLOCK.READ and store
1
Clear a storage register
Read tens of hours register
Add to storage register
Shift storage register left by 4 bits (1 digit)
Read units of hours register
Add to storage register
Shift storage register left by 4 bits (1 digit)
Read tens of minutes register
Add to storage register

Figure 3.16: Flowchart of the routine GET.TIME


Shift storage register left by 4 bits (1 digit)
Read units of minutes register
Compare with the value read before
Same value?
n
y
1
Recover processor entry state
Return to the calling routine

that was stored at the beginning of the routine. If these are different then the read was performed during a clock transition on the registers of interest and the read is reperformed. If the value had not changed then the time is placed in the I/O register, d7, and the registers are returned to their entry state. Execution then returns to the calling routine.

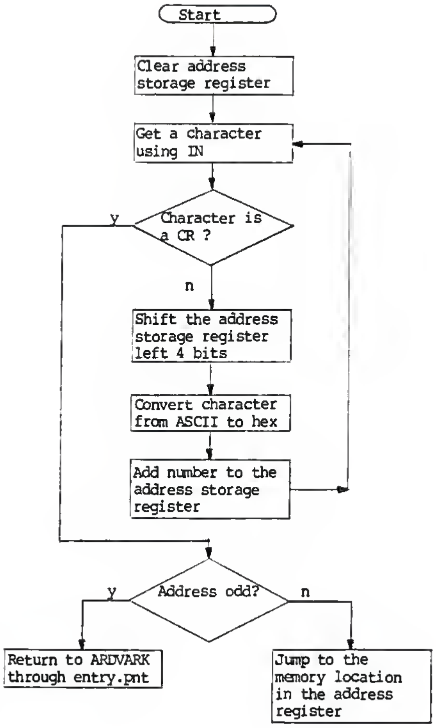
GO:

The routine GO provides a means for executing programs stored in the harmonic analyzers memory. The memory address is specified by the user on the host computer. A flow chart of GO is shown in Figure 3.17

The operation of the routine GO is fairly straightforward. The address storage register, d0, is cleared. A character is then fetched from the primary serial port. If the character is a carriage return then the program checks the address that is stored to see if it is even. If it is the program jumps to that memory location. If the address is odd then GO jumps to the entry point of ARDVARK.

If the character received is not a carriage return (CR) then it is converted from ASCII to a hexidecimal digit by the routine ASCII.HEX. The storage register is multiplied by 16 to shift the value in it left one hex digit. The just

Figure 3.17: Flowchart of the routine GO


Start
Clear address storage register
Get a character using IN
Character is a CR ?
y
n
Shift the address storage register left 4 bits
Convert character from ASCII to hex
Add number to the address storage register
Address odd?
y
n
Return to ARDVARK through entry.pnt
Jump to the memory location in the address register

received number is then added into the opening caused by the shift. Another character is fetched and the process repeats until a CR is received.

HEX.ASCII:

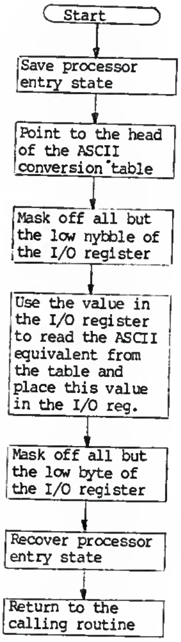
HEX.ASCII performs the conversion of a hex digit to its ASCII equivalent representation. The hexadecimal digit is passed in the low nybble of d7. The returned ASCII character is in the low byte of d7. Other than d7 no register contents are altered by this routine. Figure 3.18 shows a diagram of the HEX.ASCII routine.

On entry to HEX.ASCII the registers used by the routine are saved on the stack. A pointer to the conversion table is then initialized. The register d7 is then masked so that all but the low four bits are cleared. The resulting number is then used as the offset into the conversion table. For each hex number its ASCII equivalent is stored as an entry in the table. The entry pointed to is then retrieved and placed in d7. The registers altered in the routine are then recovered from the stack where they were saved on entry. Execution then returns to the calling routine.

HEX.CHECK:

HEX.CHECK examines an ASCII character and determines if

Figure 3.18: Flowchart of the routine HEX.ASCII


Start
Save processor entry state
Point to the head of the ASCII conversion table
Mask off all but the low nybble of the I/O register
Use the value in the I/O register to read the ASCII equivalent from the table and place this value in the I/O reg.
Mask off all but the low byte of the I/O register
Recover processor entry state
Return to the calling routine

it is the equivalent of a hex number. If it is an equivalent then the carry bit is cleared, otherwise it is set. The character to be checked is passed to the routine in register d7. No registers are altered by this routine. The flowchart for HEX.CHECK is shown in Figure 3.19.

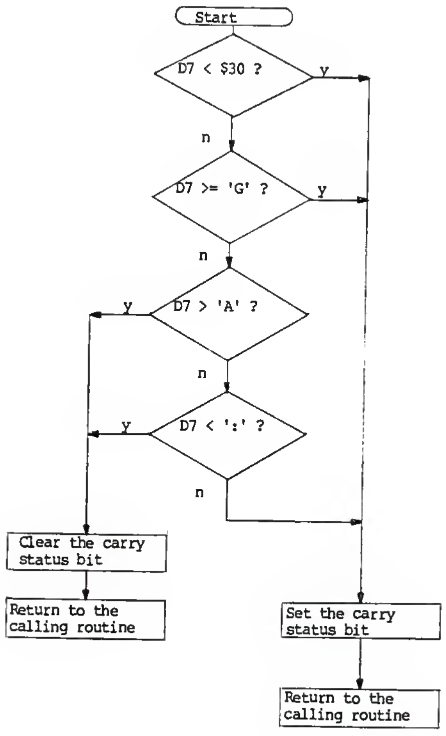
The character is first compared with ASCII zero, $30. If it is less than $30 then the carry is set and the program returns to the calling routine. If greater than $30 then it is compared with ASCII "G", or $47. If it is greater than $47 then the carry is set and the routine returns to the calling routine.

If the character is between "0" and "F" it can still be a number of nonhexadecimal equivalent ASCII characters. An additional check is then done to see if the character is less than ":", $3A, or greater than "@", $40. If this is the case then the carry is set, otherwise the carry is cleared to signal a nonhex equivalent character. Execution then returns to the calling routine.

IN:

The routine IN controls input to the harmonic analyzer via the primary serial port. It echoes back whatever character it receives to show that the character has been received. The character received is returned in register d7.

Figure 3.19: Flowchart fo the routine HEX.CHECK


Start
D7 < $30 ?
y
n
D7 >= 'G' ?
y
n
D7 > 'A' ?
y
n
D7 < ':' ?
y
n
Clear the carry status bit
Return to the calling routine
Set the carry status bit
Return to the calling routine

No other registers are altered by this routine. Figure 3.20 shows a flowchart of IN.

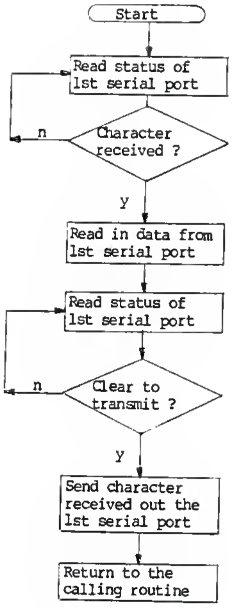
The first action of the routine IN is to read the status of the primary serial port. If no character has been received then the status continues to be read until one is received. The character is then read in from the serial port. The status of the port is read again to determine whether the transmit register is ready. If not the status is read until it is. The received character is then echoed out the primary serial port. Execution then returns to the calling routine.

INITIALIZE:

INITIALIZE is a routine to initialize the harmonic analyzer. It sets the harmonic analyzer's real time clock and initializes the system variables. No registers are altered by this routine. A flowchart of INITIALIZE is shown in Figure 3.21.

INITIALIZE begins by saving the contents of the processor registers that are changed by the routine on the stack. It then reads a character string that gives the time to set the clock to. The system variables are then initialized. These include the data pointer, sample counter and sampling enabled flag. The rest of the routine configures and sets

Figure 3.20: Flowchart of the routine IN



Start
Read status of 1st serial port
n
Character received ?
y
Read in data from 1st serial port
Read status of 1st serial port
n
Clear to transmit ?
y
Send character received out the 1st serial port
Return to the calling routine

Figure 3.21: Flowchart of the routine INITIALIZE

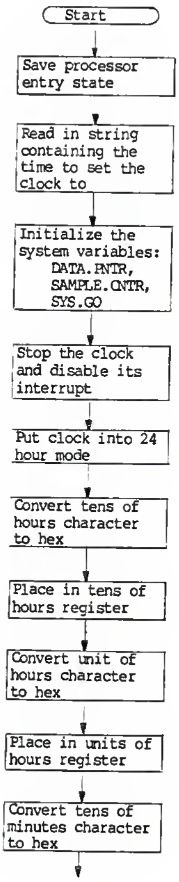
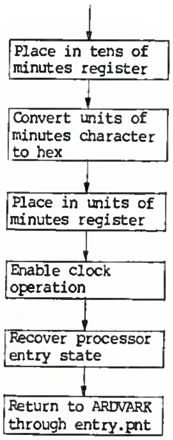

Start
Save processor entry state
Read in string containing the time to set the clock to
Initialize the system variables: DATA.PNTR, SAMPLE.CNTR, SYS.GO
Stop the clock and disable its interrupt
Put clock into 24 hour mode
Convert tens of hours character to hex
Place in tens of hours register
Convert unit of hours character to hex
Place in units of hours register
Convert tens of minutes character to hex

Figure 3.21: Flowchart of the routine INITIALIZE


Place in tens of minutes register
Convert units of minutes character to hex
Place in units of minutes register
Enable clock operation
Recover processor entry state
Return to ARDVARK through entry.pnt

the clock.

The clock is set using the routine CLOCK.WRITE. The clock is first halted with its interrupts masked. The clock is then put into the twenty four hour mode with the interrupts disabled. The first character in the time string is then converted to hexidecimal using ASCII.HEX. This number is then written to the tens of hours register of the harmonic analyzer's clock. The second character of the string is then converted and placed in the units of hours register of the clock. The third and fourth characters of the time string are converted and then placed in the tens and units of minutes registers respectively. The clock is then started. This concludes the functions of the routine and the return sequence begins.

The registers altered by this routine are placed in their original state. Program flow then returns to ARDVARK at the entry node entry.pnt. Since program flow does not return to the calling routine this is not a true subroutine.

MEMORY:

MEMORY is the routine that controls the examination and alteration of memory locations. It is not a subroutine as it is called by a normal jump and returns to ARDVARK through the entry point entry.pnt (see listings in Appendix B).

Figure 3.22 shows the flowchart of memory.

Memory functions in the following manner. It first clears a register for address storage. A character is then fetched from the primary serial port. If it is a carriage return (CR) then execution proceeds by testing the received address. If the character is not a CR then it is converted to hexidecimal using ASCII.HEX. The address storage register is shifted left by four bits to cause a shift of the stored address one hex digit to the left. The digit just received is then added in as the least significant digit. This continues until a CR is received.

The received address is then tested. It is first checked to see if the address is in the range of the actual physical memory of the harmonic analyzer. If not an error code is sent out the primary serial port and the execution returns to ARDVARK through entry.pnt. If the address is in range then it is checked to see if it is an even address. If not even then an error code is sent and execution returns to ARDVARK. If it is even then the code for no error is transmitted.

MEMORY now displays the data at an address. First the address is sent in ASCII coded hexidecimal digits. The data at the address is then fetched and sent in the same manner.

Figure 3.22: Flowchart of the routine MEMORY

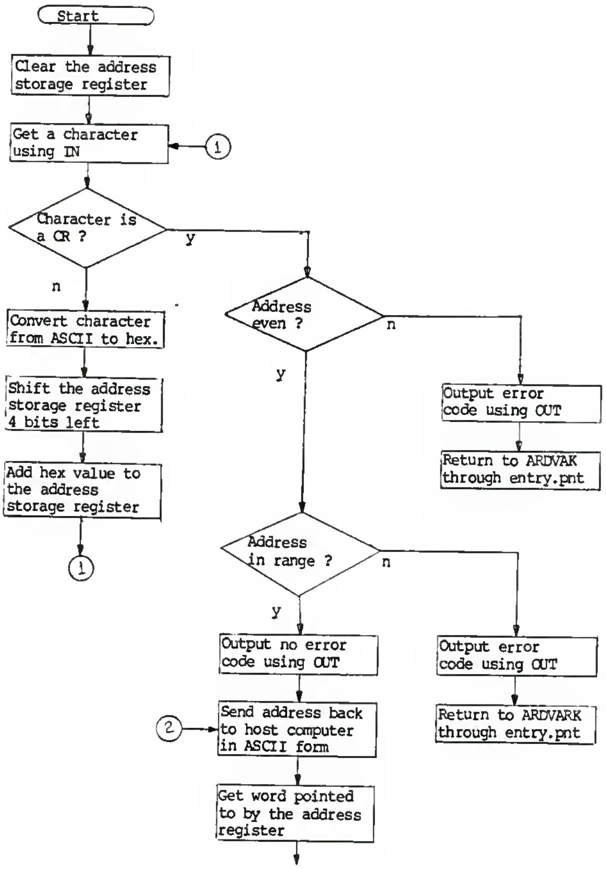

Start
Clear the address storage register
Get a character using IN
1
Character is a CR ?
y
n
Convert character from ASCII to hex.
Shift the address storage register 4 bits left
Add hex value to the address storage register
1
Address even ?
n
y
Output error code using OUT
Return to ARDVAK through entry.pnt
Address in range ?
n
y
Output no error code using OUT
Output error code using OUT
Return to ARDVARK through entry.pnt
2
Send address back to host computer in ASCII form
Get word pointed to by the address register

Figure 3.22: Flowchart of the routine MEMORY

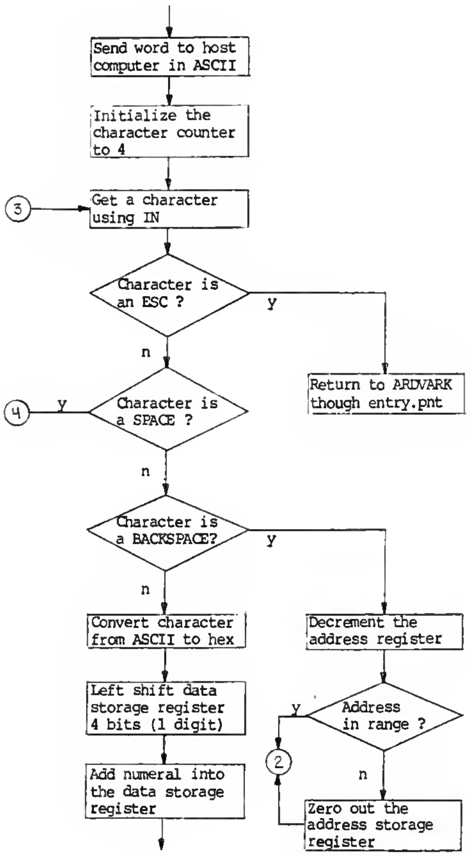
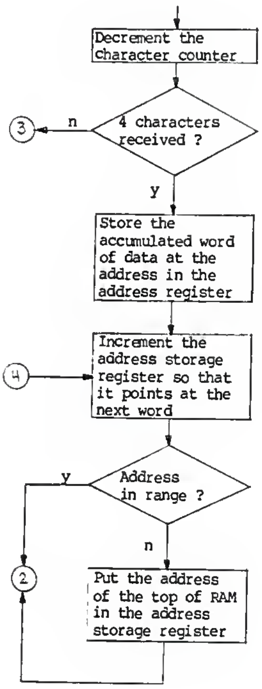

Send word to host computer in ASCII
Initialize the character counter to 4
3
Get a character using IN
Character is an ESC ?
y
n
Return to ARDVARK though entry.pnt
4
y
Character is a SPACE ?
n
Character is a BACKSPACE?
y
n
Convert character from ASCII to hex
Decrement the address register
Left shift data storage register 4 bits (1 digit)
y
Address in range ?
2
n
Add numeral into the data storage register
Zero out the address storage register

Figure 3.22: Flowchart of the routine MEMORY

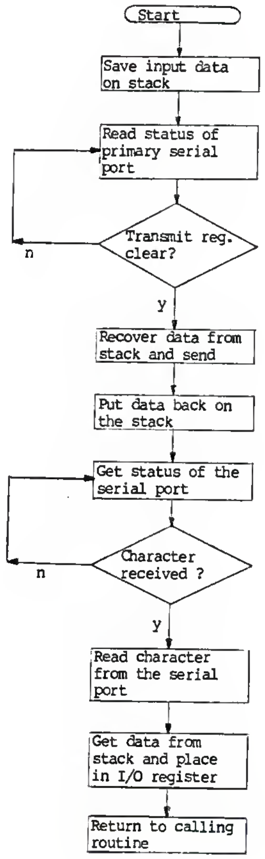

Decrement the character counter
4 characters received ?
n
3
y
Store the accumulated word of data at the address in the address register
Increment the address storage register so that it points at the next word
4
Address in range ?
y
n
2
Put the address of the top of RAM in the address storage register

The next section of memory allows the alteration of the displayed memory location. It also allows the user to increment or decrement the address without altering the contents of the present location. This section is set up so that when four ASCII coded hexidecimal digits are received the value they represent is stored in the memory location addressed. If a space or back space is received at any point in this sequence the address is incremented or decremented, respectively. If an escape character, $1B, is received then execution is returned to ARDVARK through entry.pnt. When the address is incremented or decremented it is checked to see if it still is in the memory range of the harmonic analyzer. If the address is not in range it is set to $0000 or the top of RAM depending on whether the fault occurred while the address was decremented or incremented, respectively.

OUT:

The routine OUT is the analog of the routine IN. It controls the transmission of data out the primary serial port. Once a character has been sent out the port the routine waits for it to be echoed back. This keeps the transmitter from overrunning the receiver. The character to be sent is passed to the routine in the low byte of d7. No register contents are altered by this routine. Figure 3.23 shows a flowchart detailing the operation of OUT.

Figure 3.23: Flowchart of the routine OUT


Start
Save input data on stack
Read status of primary serial port
Transmit reg. clear?
n
y
Recover data from stack and send
Put data back on the stack
Get status of the serial port
Character received ?
n
y
Read character from the serial port
Get data from stack and place in I/O register
Return to calling routine

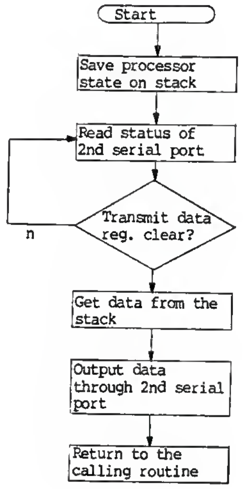
OUT begins operation by saving the character to send on the stack. It then reads in the status of the primary serial port. If the transmit data register is not ready then status is read until it is. When the port is ready to transmit, the data is recovered from the stack and sent out the port. The data is saved once again on the stack. The routine now waits for an echo. This prevents data from being sent out faster than it can be received. The status of the port is then read. If no character has been received then the status is read until one is. Once a character has been received the port is read. The data is recovered from the stack again and execution returns to the calling routine.

PUTCHAR:

PUTCHAR is a I/O primative. It transmits a character out the secondary serial port. Unlike the routine OUT, PUTCHAR does not wait for an echo to be returned from the receiver. The character to be transmitted is passed in the low byte of d7. No registers are altered by this routine. Figure 3.24 shows the operation of PUTCHAR.

PUTCHAR operates in the following manner. It places the data to be sent on the stack. The status of the second serial port is then read to see if the transmit register is ready. If the transmit register is not ready then the status

Figure 3.24: Flowchart of the routine PUTCHAR


Start
Save processor state on stack
Read status of 2nd serial port
Transmit data reg. clear?
n
Get data from the stack
Output data through 2nd serial port
Return to the calling routine

is read until it is. PUTCHAR gets the data from the stack and sends it out the secondary serial port. Execution then returns to the calling routine.

QK.FFT:

The routine QK.FFT samples the voltage and curent waveforms for one cycle, performs an FFT on them and sends the results to the host computer. This allows a quick check of line conditions as well as being a method of testing the operation of the harmonic analyzer. No registers are changed by this routine. Figure 3.25 shows the operation of QK.FFT.

The following description details the operation of QK.FFT. QK.FFT first saves the contents of registers that are altered by this routine so that they can be recovered when exiting the routine. It then samples the data using the routine GET.DATA which samples the voltage and current waveforms for one cycle. It performs an FFT on the voltage and current data by calling the routine FFT. The data is then sent out the primary serial port. The data is sent by placing a pointer to the beginning of the voltage data. The byte pointed to is transmitted via the routine OUT. The pointer is incremented to point to the next byte. Since the current data follows immediately after the voltage data (see the memory map in Appendix D) this can continue until the

Figure 3.25: Flowchart for the routine QK.FFT

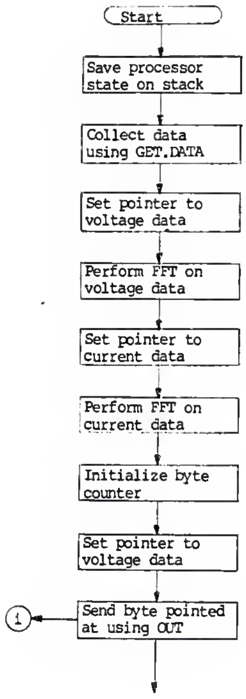
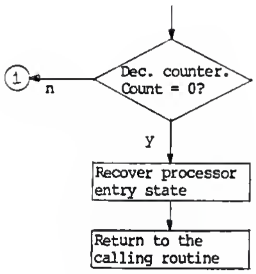

Start
Save processor state on stack
Collect data using GET.DATA
Set pointer to voltage data
Perform FFT on voltage data
Set pointer to current data
Perform FFT on current data
Initialize byte counter
Set pointer to voltage data
Send byte pointed at using OUT
1

Figure 3.25: Flowchart for the routine QK.FFT



end of the current data. Transmission ends at the end of the block of current data.

This concludes the functions of QK.FFT. The program then goes into the return phase. The registers that were altered in this routine are recovered from the stack and placed in their original condition. Execution then returns to ARDVARK through the entry node entry.pnt. For this reason QK.FFT is not a true subroutine.

SAMPLE:

SAMPLE controls the acquisition and storage of data by the harmonic analyzer. It checks to see if sampling should occur. If it should then the data is collected, an FFT performed and the results are stored in memory. The time is passed to this routine in the low word of d7. Figure 3.26 shows a flow chart of SAMPLE.

SAMPLE first saves the contents of registers that will be altered on the stack. The variable SYS.GO is then checked to see if sampling is enabled. This is followed by a check of the sample counter. If sampling is not enabled or the sample counter is equal to 168 then SAMPLE goes to the exit phase, otherwise it samples the voltage and current wave-forms by calling the routine GET.DATA. It then performs an FFT on the voltage data followed by a FFT on the current

Figure 3.26: Flowchart of the routine SAMPLE

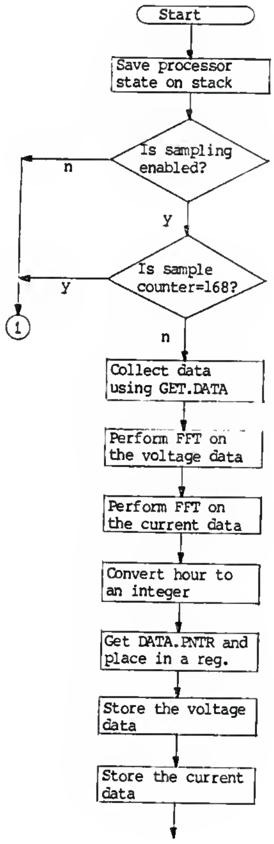
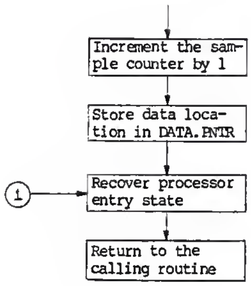

Start
Save processor state on stack
Is sampling enabled?
n
y
Is sample counter=168?
y
1
n
Collect data using GET.DATA
Perform FFT on the voltage data
Perform FFT on the current data
Convert hour to an integer
Get DATA.PNTR and place in a reg.
Store the voltage data
Store the current data

Figure 3.26: Flowchart of the routine SAMPLE


Increment the sample counter by 1
Store data location in DATA.PNTR
1
Recover processor entry state
Return to the calling routine

data by calling the routine FFT.

The clock data that was passed to SAMPLE is then converted from a set of BCD numbers representing hours and minutes to a hexidecimal number representing just the hours. This is done by shifting the time data right eight bits. This puts the BCD coded tens and units of hours in the low byte of a register. The nybble for the tens of hours is multiplied by ten to produce its hexidecimal equivalent. The nybble representing the units of hours is then added to this. The results in a hexidecimal word length integer that gives the hour when the data was taken.

SAMPLE stores all the data next. The data pointer is fetched from the system variable data.pntr. The time integer is then stored followed by the voltage data and current data. Not all of the harmonics are stored. The even harmonics from 24 to 62 are left aside. These harmonics are generally negligible and omitting them allowed longer data collection runs without increasing the amount of memory required. The new value of the data pointer resulting from the storage of the data is placed back in the variable data.pntr.

The contents of the altered registers are returned to their entry state by recovering these values from the stack where they were saved. Program flow then returns to ARDVARK through the node entry.pnt.

SEND.DATA:

SEND.DATA sends out the data accumulated by the harmonic analyzer to the host computer. The data is sent out as word length integers in Intel format. The least significant byte of the integer is sent first followed by the most significant byte. No registers are altered by this routine. Figure 3.27 shows the operation of SEND.DATA in flowchart form. The following paragraphs detail the functioning of SEND.DATA.

SEND.DATA sends the data as packets of related information. Each sample block, or data packet, is made up of the hour and the harmonic information for that hour for both the current and voltage waveforms. The time is a word length integer that gives the number of the hour the data was taken. The harmonics stored are 0-23 and 25-63 odd. The harmonic information itself is in integer form. Each harmonic takes up 4 bytes of storage, a word integer for the real part and a word integer for the imaginary part. Figure 3.28 shows the data block in graphic form. The length of each block is 177 words.

A checksum is appended to the end of this packet for

Figure 3.27: Flowchart of the routine SEND.DATA

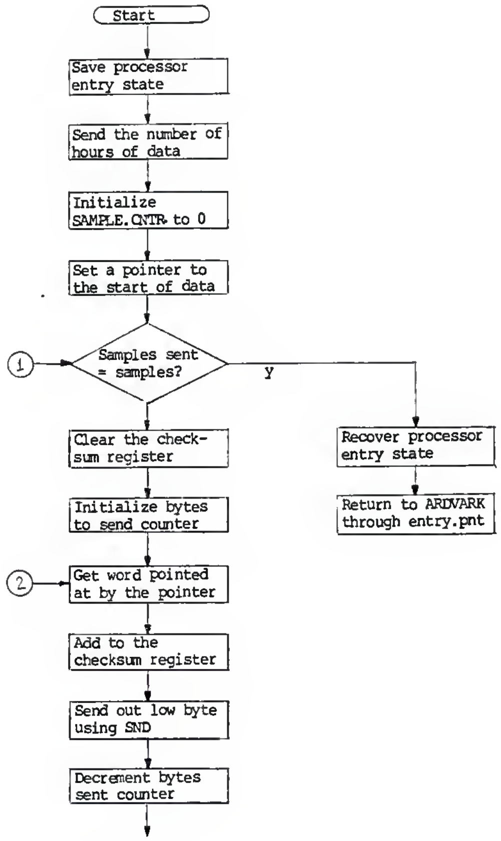
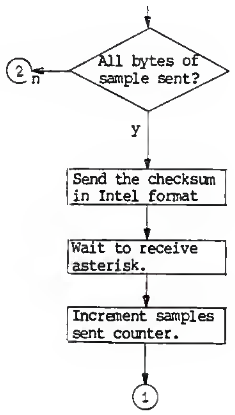
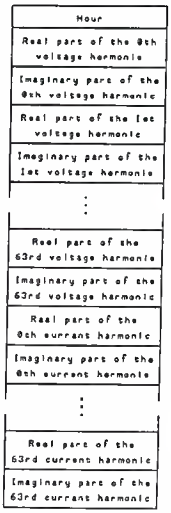

Start
Save processor entry state
Send the number of hours of data
Initialize SAMPLE.CNTR to 0
Set a pointer to the start of data
1
Samples sent = samples?
y
Clear the check-sum register
Initialize bytes to send counter
2
Get word pointed at by the pointer
Add to the checksum register
Send out low byte using SND
Decrement bytes sent counter
Recover processor entry state
Return to ARDVARK through entry.pnt

Figure 3.27: Flowchart of the routine SEND.DATA


All bytes of sample sent?
n
2
y
Send the checksum in Intel format
Wait to receive asterisk.
Increment samples sent counter.
1

error checking purposes. The checksum is a word length integer. It is computed by adding each byte in the data packet modulo 65535. The transmission of the checksum is the final action in sending the data packet.

The I/O performed in this routine is nonstandard for the primary serial port. Instead of using the standard routines for serial I/O, IN and OUT, SEND.DATA uses the routines SND and RCV. These routines do not perform the echoing that IN and OUT do. This enables the data to be sent at a faster rate. The listings for SND and RCV are included



Hour
Real part of the 0th voltage harmonic
Imaginary part of the 0th voltage harmonic
Real part of the 1st voltage harmonic
Imaginary part of the 1st voltage harmonic
Real part of the 63rd voltage harmonic
Imaginary part of the 63rd voltage harmonic
Real part of the 0th current harmonic
Imaginary part of the 0th current harmonic
Real part of the 63rd current harmonic
Imaginary part of the 63rd current harmonic

Figure 3.28: Data storage format for each hour of data in the harmonic analyzer and format of data transmission packet.

with those of SEND.DATA in Appendix B. The functional descriptions of SND and RCV are the same as those for PUTCHAR and GETCHAR except that SND and RCV deal with the primary serial port instead of the second.

SEND.DATA operates in the following manner. SEND.DATA first saves the registers that are altered in the routine on the stack. The variable sample.cntr is read and sent out as an integer. It then clears the counter for the number of samples sent. The data pointer is then set to the beginning of the data storage block of memory. SEND.DATA then enters a cyclic state to send out the samples.

Each cycle begins with a check of the number of samples sent. If this equals the number of samples stored in the harmonic analyzer then SEND.DATA enters the exit mode covered below. If the number of samples sent does not equal the number of samples stored then the following actions occur. The checksum register is cleared and the 'words to send' counter is initialized. The word of data pointed to by the data pointer is fetched and the data pointer is incremented. This value is added into the checksum register. The low byte is sent followed by the high byte of the word. This is done using the routine snd. If the end of the data block is not reached then the next word is fetched and action is continued until the end of the data block is reached. When the end occurs the accumulated checksum of the block is sent out, least significant byte first. The harmonic analyzer now waits for the reception of an asterisk to signal that the packet has been received. When this is received the cycle begins again until all the samples have been sent.

SEND.DATA then recovers the contents of registers that were changed by the routine from the stack. Execution then returns to ARDVARK through entry.pnt.

SEND.TIME:

The routine SEND.TIME sends the time read from the harmonic analyzers clock out the primary serial port in ASCII coded form. The time is sent as four characters representing hours and minutes. No registers are changed by this routine. Figure 3.29 shows the functioning of this routine.

As in most of the ARDVARK subroutines SEND.TIME places the contents of the registers that will be altered by the routine on the stack so the original contents can be recovered when the routine is concluded. SEND.TIME then reads the data from the clock using the routine GET.TIME and saves it in a storage register.

The time is then copied back into the I/O register and shifted right. This moves the most significant nybble in the

Figure 3.29: Flowchart for the routine SEND.TIME

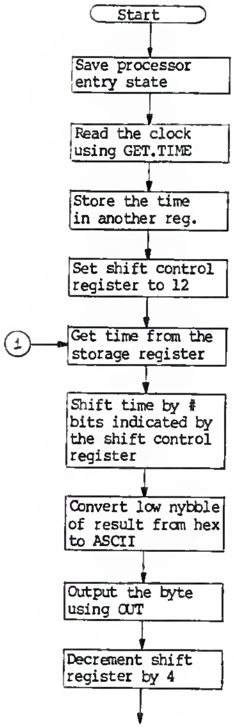
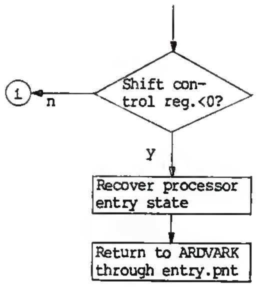

Start
Save processor entry state
Read the clock using GET.TIME
Store the time in another reg.
Set shift control register to 12
1
Get time from the storage register
Shift time by # bits indicated by the shift control register
Convert low nybble of result from hex to ASCII
Output the byte using OUT
Decrement shift register by 4

Figure 3.29: Flowchart for the routine SEND.TIME


Shift control reg.<0?
n
1
y
Recover processor entry state
Return to ARDVARK through entry.pnt

place of the least significant nybble. This is then converted to ASCII and sent out using OUT. The shift control register is decremented so that the next most significant nybble will be shifted to the low nybble location. This cycle continues until all four nybbles have been sent.

SEND.TIME then recovers the entry state of any registers changed in the routine. Execution returns to ARDVARK through the entry point entry.pnt.

TRACE:

TRACE allows the user of the harmonic analyzer to single step through programs. It requires the use of a terminal attached to the secondary serial port. It is not really a subroutine attached to ARDVARK. Instead it is an exception routine that is called every time an instruction is executed. This requires that the trace enable bit of the microprocessor be active. This is taken care of in TRACE itself.

Using TRACE is very simple. In order to enter the trace mode a "T" is sent to the harmonic analyzer via the primary serial port. This causes the execution of a trap to the trace exception. This causes the contents of all the processor registers to be displayed on the screen. The message "trace off" is then displayed. A character must be entered to continue. If the character is a carriage return (CR) then the state of the trace bit remains the same. If any other character is entered then the trace bit is toggled. When first entering the TRACE routine enter any character but CR to get into the "trace on" mode. Once a character has been entered the message "pc:" will appear on the next line. This allows the alteration of the program counter. Type the address of the next statement to execute. If the first character entered is a CR then the current value of the program counter is used. No checking is made for odd addresses. Once the address has been entered the next instruction is executed. If the trace bit is enabled then the trace is returned to and the results are the same as above except that the message displayed after the registers is "trace on." If the trace bit is off execution continues in the normal fashion.

The exception trace, shown in Figure 3.30, operates in the following way. All the processor registers are saved onto the stack. A pointer is set to the top of the stack. The registers are then copied into the I/O register, d7, and displayed using the routine DISP.REG. A label is displayed for the register using the routine DISP.STRING. This continues until all the registers have been displayed.

The next section of trace controls the toggling of the

Figure 3.30: Flowchart for the exception TRACE

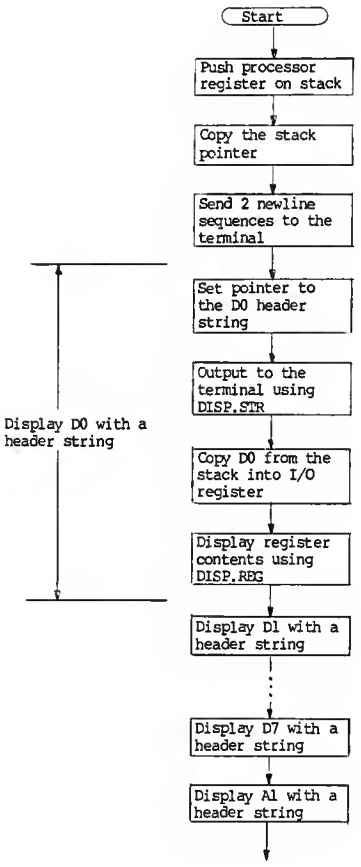
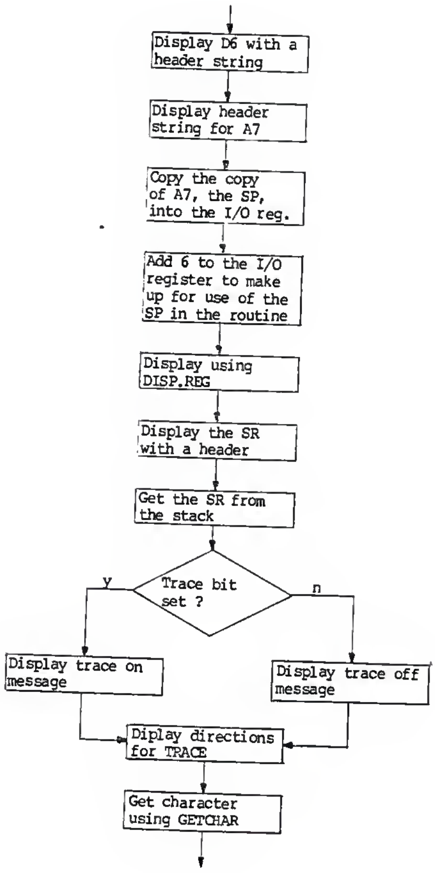

Start
Push processor register on stack
Copy the stack pointer
Send 2 newline sequences to the terminal
Set pointer to the D0 header string
Output to the terminal using DISP.STR
Copy D0 from the stack into I/O register
Display register contents using DISP.REG
Display D0 with a header string
Display D1 with a header string
Display D7 with a header string
Display A1 with a header string

Figure 3.30: Flowchart for the exception TRACE



Display D6 with a header string
Display header string for A7
Copy the copy of A7, the SP, into the I/O reg.
Add 6 to the I/O register to make up for use of the SP in the routine
Display using DISP.REG
Display the SR with a header
Get the SR from the stack
Trace bit set ?
y
n
Display trace on message
Display trace off message
Diplay directions for TRACE
Get character using GETCHAR

Figure 3.30: Flowchart for the exception TRACE

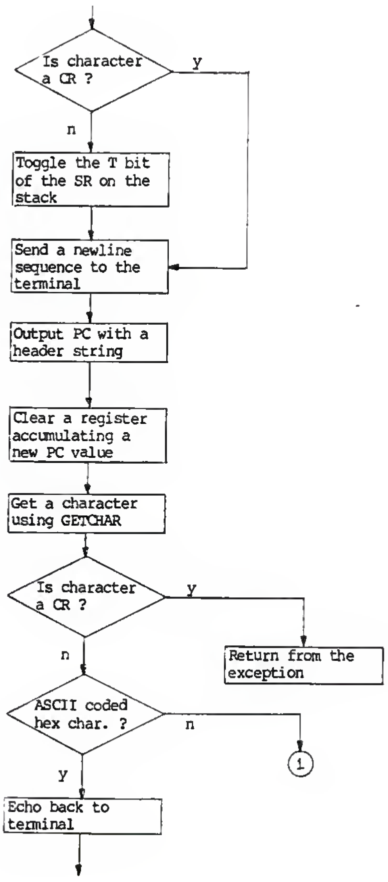
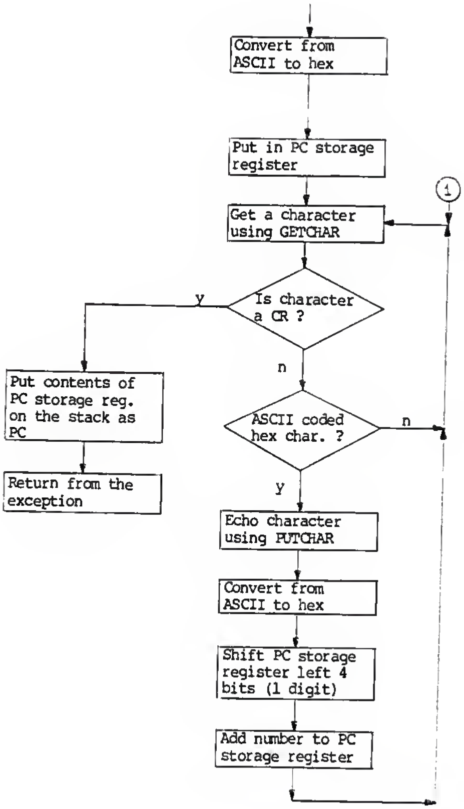

Is character a CR ?
y
n
Toggle the T bit of the SR on the stack
Send a newline sequence to the terminal
Output PC with a header string
Clear a register accumulating a new PC value
Get a character using GETCHAR
Is character a CR ?
y
Return from the exception
n
ASCII coded hex char. ?
n
1
y
Echo back to terminal

Figure 3.30: Flowchart for the exception TRACE



Convert from ASCII to hex
Put in PC storage register
1
Get a character using GETCHAR
Is character a CR ?
y
n
Put contents of PC storage reg. on the stack as PC
Return from the exception
ASCII coded hex char. ?
n
y
Echo character using PUTCHAR
Convert from ASCII to hex
Shift PC storage register left 4 bits (1 digit)
Add number to PC storage register

trace bit. First the status of the trace bit is checked and depending on the state the message "trace on" or "trace off" is displayed using the routine DISP.STRING. Next a character is fetched from the secondary serial port using GETCHAR. If the character returned is not a CR then the trace bit is toggled in the copy of the status register stored on the stack.

The next section of TRACE deals with modifying the program counter. A new line sequence is sent out using the routine DISP.STRING. The label string for the program counter is displayed using DISP.STRING. A character is fetched using the routine GETCHAR. If this character is a CR then execution proceeds to the return phase of TRACE. If it is not a CR then it is checked to see if it is hex. If so it is echoed back to the screen, converted to hexidecimal and added into the address storage register. If the character is not a hexadecimal equivalent then no echo is sent to the terminal and a new character is fetched. Each succeeding character is given a similar treatment except that the storage register is shifted four bits left before the succeeding values are added in. This clears the low nybble and places the value just received in the least significant position of the address. This continues until a CR is received. Trace then goes into the return phase.

The return phase of TRACE clears two more lines on the screen using DISP.STRING. The registers are then read back from the stack. A return from exception is then executed and TRACE goes back to execute the instruction pointed at by the program counter. If the trace bit is set this causes a trace exception to be executed again. If the trace bit is clear then execution proceeds in the usual fashion.

CHAPTER 4 : TESTING

Testing Procedure

A testing procedure for the harmonic analyzer was developed to evaluate its performance. Due to the fact that spectrally pure 5A, 120V sources are difficult to come by, the testing procedure used bypassed the sensors for the harmonic analyzer. 5A, 120V sources were used to test the sensors in a more general sort of way but the input signals were not predictable enough to be considered adequate tests. The testing procedure used is outlined in the following paragraphs.

The tests were performed with the following physical setup. The input waveforms were obtained from an HP 3314A synthesizer in the square wave and triangle wave modes. The input signal is fed into the input of the voltage sensor's sample and hold. This bypasses the resistive divider that scales the actual line voltages down. This should not significantly change the results of the test.

The harmonic analyzer is then connected to the Zenith Z-171 microcomputer which is used to monitor the harmonic analyzer's operation and gather data. The Z-171 is running SETUP, the harmonic analyzer/Z-171 communications program.

The Z-171 is then used to cause the harmonic analyzer to perform an FFT on the waveform and send the data on the spectrum to the Z-171.

The two tests performed are the square wave test and the triangle wave test. For the square wave test a 10.0 Vpp 60.0 Hz square wave with no DC offset is used as the input signal. In the triangle wave test a 10.0 Vpp, 60.0 Hz triangle wave is used as the input signal. The output data on the spectrum of the input signals are then compared with ideal values for the appropriate waveform's spectrum.

The results of both tests for five trials each, are shown in Table 4.1 and Table 4.2. The values shown are in percentages of the amplitude of the fundamental frequency, 60.0 Hz. The sampled data was averaged and the maximum negative and positive deviations were noted. This was performed for the first 14 harmonics of 60.0 Hz in both the square wave and the triangle wave tests.

As can be seen from the tables of results the harmonic analyzer is generally accurate to about 0.1%. The error increases compared to the magnitude of the harmonic as the order of the harmonic increases. This is due to the quantization noise inherent in the analog to digital conversion. The signal to noise ratios for lower level signals are then less. In the waveforms used in the test the amplitude of the various harmonics decreases with frequency, so the error increases.

One of the design goals for the harmonic analyzer project was to achieve an accuracy of better than 1% preferably to within 0.1%. Based on the bench mark tests this design goal was met.

Table 4.1 Results of Square Wave Test

harmonic	ideal % of fundamental	avg. % of fundamental	+ deviation	- deviation
1	100	100	0	0
2	0	0	0	0
3	33.3	33.4	0.22	-0.22
4	0	0	0	0
5	20.0	20.06	0.09	-0.143
6	0	0	0	0
7	14.3	14.34	0.053	-0.064
8	0	0	0	0
9	11.1	11.23	0.067	-0.064
10	0	0.016	0.062	-0.016
11	9.09	9.23	0.061	-0.061
12	0	0	0	0
13	7.69	7.84	0.055	-0.060
14	0	0	0	0

Table 4.2 Results of Triangle Wave Test

harmonic	ideal % of fundamental	avg. % of fundamental	+ deviation	- deviation
1	100	100	0	0
2	0	0.184	0.0025	-0.0046
3	11.11	11.23	0.207	-0.292
4	0	0	0	0
5	4.00	4.01	0.0603	-0.820
6	0	0	0	0
7	2.04	2.09	0.046	-0.033
8	0	0	0	0
9	1.23	1.22	0.012	-0.0097
10	0	0	0	0
11	0.83	0.846	0.034	-0.028
12	0	0	0	0
13	0.58	0.584	0.011	-0.024
14	0	0	0	0

CHAPTER 5 : FURTHER CONSIDERATIONS AND CONCLUSIONS

Further Considerations

At this point a number of future considerations and improvements can be mentioned. These can be grouped into the areas of basic system changes, hardware improvements and software improvements.

The first basic system change is to add the subsystems necessary to monitor three phases of voltage and current. This would enable the analyzer to monitor the relations between harmonics on different phases. Also in order to avoid aliasing problems an anti-aliasing filter should be added. The software system could be modified by adding some self test diagnostic routines that would test some of the harmonic analyzers systems.

There are numerous hardware improvements that could be done. The most important is to redesign the board to eliminate the noise problems in the digital section. A future design could possibly use a multilayer board with a ground plane in conjunction with better layout to reduce noise. The microprocessor clock could be seperated from the sampling clock system so the microprocessor could be run at a higher speed.

Many of the chips used in the current version of the harmonic analyzer could be replaced by newer more efficient and powerful chips. The 6800 style timing used to interface with these slower peripherals could be eliminated with the newer faster chips. Some of the improvements in this area are to replace the RAM, serial ports and parallel port. The use of 43256 32kx8 RAM's, 2681 dual serial ports would produce lower chip counts and more functionality than the present chips.

The use of custom chips could significantly reduce the chip count of the harmonic analyzer. The decoding logic used for chip selects is presently implemented by discrete chips. The use of programmable array logic (PAL) chips could reduce the number of chips needed for these functions to about two. Additionally PAL's could be used to control the action of the sampling port.

The software changes mainly deal with improving the signal processing routines. The present FFT could be used to perform a Walsh technique analysis of the data. In this the incoming data would be sampled over a number of cycles, n, the FFT of this data is computed in one cycle segment and averaged. This reduces the variance of the output sequence. The reduction is walsh = fft/n. For more information on this technique see [6]. This technique can not be performed presently due to lack of buffer memory on the harmonic analyzer.

These improvements would increase the overall system performance. They should increase the functionality to board space ratio and provide a faster more accurate and reliable system.

Conclusions

A low cost spectrum analyzer for use in measuring power system harmonics has been developed. The analyzer will measure current and voltage harmonics each hour for one week. The data acquired by the analyzer can be gathered by an MS-DOS compatible computer. The accuracy of the analyzer compares to what was expected based on benchmark tests.

The harmonic analyzer does have a number of problems. The current hardware model appears to have some noise problems in the digital section. This reduces system reliability. It did not affect the project too significantly but would not be acceptable in a commercial machine. Also there exists the possibility of aliasing in sampling the input waveforms. This would decrease the accuracy of the analyzer if it were significant.

This document discussed the hardware and software systems that make up the harmonic analyzer. The hardware was discussed at the component level. Some of the decisions that were made in deciding on the form of the hardware are also discussed. The software system was examined on a routine by routine basis at the flowchart level to explain the methodology used.

APPENDIX A

Parts list and memory map for the Harmonic Analyzer. The schematics for the Harmonic Analyzer are included in Chapter 2.

Parts List for the Harmonic Analyzer

Integrated Circuits

U1	: 741 operational amplifier
U2	: AD524 precision instrumentation amplifier Analog Devices
U3, U4	: SMP-10 Sample and Hold Amplifiers Precision Monolithics Inc.
U5	: MUX-08 eight channel multiplexer Precision Monolithics Inc.
U6	: LM310 voltage follower National Semiconductor
U7	: 4.9152 Mhz TTL level clock generator
U8, U9	: 74HC74 Dual D flip/flops
U10	: CD4520 Dual 4 bit counter
U11	: CD4017 Decade Sequencer/Divider
U12	: 74HC154 4 to 16 decoder, skinny package
U13	: 68000 Microprocessor
U14, U23, U32, U33	: 74HC32 Quad two input OR
U15, U16, U18	: 74HC04 Hex inverter
U17	: 74HC175 Quad D flip/flop
U19	: 555 Timer
U20	: LM393 Dual comparator
U21, U22	: 2732 4Kx8 EPROM
U24-U31	: 6264 8Kx8 static RAM
U34	: 74HC138 3 to 8 decoder
U35	: 74HC20 Dual four input NAND

U36, U37	: 6850 Asynchronous Comm. Interface Adapter
U38	: 1489 RS-232 to TTL quad receiver
U39	: 1488 TTL to RS-232 quad transmitter
U40	: 6821 Peripheral Inteface Adapter (PIA)
U41	: 74HC08 Quad two input AND
U42	: AD574 12 bit Analog to Digital Converter Analog Devices
U43	: MM58274 Real Time Clock National Semiconductor

Resistors

R1	: 18K 0.25 W
R2	: 500 ohm multiturn trimmer potentiometer
R3, R4, R12, R20, R23	: 10K multiturn trimmer potentiometer
R5	: 205 ohm 1% 0.25W metal film
R6,R10	: 2.4K 0.25W
R7,R11	: 100K multiturn trimmer poentiometer
R8	: 90.9K 1% metal film 0.5W
R9	: 9.09K 1% metal film 0.5W
R13, R14	: 4.7K 0.25W
R15	: 2.0M 0.25W
R16	: 10.0M 0.25W
R17, R21, R22	: 1.0K 0.25W
R18,R19	: 100 ohm miltiturn trimmer pots

Capacitors

C1-C5, C7, C11, C13, C15-C20, C23-C28, C31-C40	: 0.1uF ceramic disk capacitors
C6, C8, C12, C14	: 10uF tantulum capactiors
C9,C10	: 5000pF polystyrene or teflon capacitor
C21, C22	: 1.0uF mylar capacitor
C29	: 20 pF cermaic disk
C30	: 5-30pF trimmer capacitor

Transistors

Q1	: BS170 N-channel enhancement mode MOSFET
Q2, Q3, Q4	: 2N2222A

Diodes

D1	: LM329 precision zener
D2	: 10 V transorb or two back to back 10 V zeners
D3,D6	: 3.9V 1.0W Zener
D4	: 3.6V 1.0W Zener
D5, D7	: Low to Med. power general purpose Si diode

Miscellaneous

J1, J2	: 25 pin subminature D plug
SW1	: 7 Pole DIP Switch
SW2	: 4 Pole DIP Switch
CT1	: PI-100 Hall Effect Current Sensor F. W. Bell
X1	: 32.786 Khz crystal
T1	: 10V Transorb
SIP1-SIP8	: 4.7k Common node SIP resistors Panasonic

Memory Map for the Harmonic Analyzer

	Address:	Comment:
ROM:	00 0000 - 00 03FF	68000 Vector Table
	00 0400 - 00 1FFF	Ardvark Monitor Program
	00 2000 - 00 FFFF	Reflections of ROM
I/O:	01 0000 - 01 FFFF	Memory Block with 6800 timing specifications
	01 1001	ACIA #1 Status register
	01 1003	ACIA #1 Data Register
	01 2001	ACIA #2 Status Register
	01 2003	ACIA #2 Data Register
	01 F001	PIA Data/Dir. Register A
	01 F003	PIA Control Register A
	01 F005	PIA Data/Dir. Register B
	01 F007	PIA Control Register B
RAM:	02 0000 - 02 077F	System Stack 1920 bytes
	02 0780 - 02 07FF	System Variables 128 bytes
	02 0800 - 02 09FF	Temporary storage block for voltage data
	02 0A00 - 02 0BFF	Temporary storage block for current data
	02 0C00 - 02 FFFF	Memory Block for Storage of Harmonic Data 61 Kbytes

APPENDIX B

Source code listings for ARDVARK, the harmonic analyzer's control program

```
*******************************************************************
*
*       ardvark.lnk                         {link specification}
*
*       Link specification for the harmonic analyzer's
*       monitor.
*
*       source file:    ardvark.lnk
*
*       author:         Norm Mortensen
*
*       data:           30Apr87
*
*******************************************************************

        output motorola
pc              equ.l $400
data.pntr       equ.l $20780
sample.cntr     equ.l $20784
sys.go          equ.l $20786
data.start      equ.l $20C00

        org pc
        link vectors    this module assembles absolutely at $00
        link ardvark    these link relocatably from pc up
        link in
        link out
        link getchar
        link get_data
        link putchar
        link dispreg
        link dispstr
        link asciihex
        link hexascii
        link hexcheck
        link clk_read
        link clk_wrt
        link get_time
        link fft
        link table128
        link download
        link memory
        link sample
        link init
        link snd_data
        link snd_time
        link go
        link qk_fft
        link trace      this compiles absolutely at $1000
```

```
*****************************************************************
*
*       ardvark                                       {program}
*
*       Main control routine for the harmonic analyzer.
*       It performs system reset on power up and provides
*       control for the various harmonic analyzer activities.
*
*
*       source file:    ardvark.a68
*
*       author:         Norm Mortensen
*
*       date:           28Apr87
*
*****************************************************************


*       global label definitions
        xdef entry.pnt

*       external references
        xref download
        xref get.time
        xref go
        xref in
        xref initialize
        xref memory
        xref pc
        xref putchar
        xref qk.fft
        xref send.time
        xref sample
        xref send.data
        xref set.clock
        xref sys.go


*       label definitions

stk             equu    $02fffe
sreg1           equ     $011001         acia 1 reg locations
dreg1           equ     $011003
sreg2           equ     $012001         acia 2 reg locations
dreg2           equ     $012003
a.status        equ     $01f003         a side of he pia
a.data          equ     $01f001
b.status        equ     $01f007         b side of acia
b.data          equ     $01f005
```

```
*
*       system reset and initialization
*

acia.reset      move.b  #3,sregl        master reset of acia'S
                move.b  #3,sreg2
                move.b  #$12,sregl      configure acia 1
                move.b  #2,sreg2        configure acia 2

pia.reset       move.b  #$30,a.status
                move.b  #0,a.data
                move.b  #$34,a.status
                move.b  #$00,b.status
                move.b  #$0f,b.data
                move.b  #$04,b.status
                move.b  #$0f,b.data

entry.pnt:
*               Check for char from serial port one
                move.b sregl,d7         get status
                andi #1,d7              check for char ready
                beq check.clock         if zero no char

*                   Get data and echo back to source
                    move.b dregl,d7     get data
                    andi #$ff,d7        mask off all but data
                    move d7,-(sp)       save

echo.rdy:           move.b sregl,d7     check for empty tx reg
                    andi #2,d7
                    beq echo.rdy
                    move (sp)+,d7
                    move.b d7,dregl     send echo
                    bsr putchar         display on terminal

*                   Decode the input character and route
*                   flow accordingly

                    cmpi.b #'M',d7      if m then memory
                    beq memory          examine/alter routine
                    cmpi.b #'m',d7
                    beq memory

                    cmpi.b #'G',d7      if x then send off
                    beq send.data       data routine
                    cmpi.b #'g',d7
                    beq send.data

                    cmpi.b #'X',d7      if g then program
                    beq go              execution routine
```

```
                  cmpi.b #'x',d7
                  beq go

                  cmpi.b #'D',d7      if d then the file
                  beq download        downloading routine
                  cmpi.b #'d',d7
                  beq download


                  cmpi.b #'T',d7      if t then the instruction
                  beq tracer          trace routine
                  cmpi.b #'t',d7
                  beq tracer

                  cmpi.b #'F',d7
                  beq qk.fft          get data and run an fft
                  cmpi.b #'f',d7      on it
                  beq qk.fft

                  cmpi.b #'I',d7      harmonic analyzer init
                  beq initialize      routine
                  cmpi.b #'i',d7
                  beq initialize

                  cmpi.b #'B',d7      set sys flag to okay
                  beq enable.sampling sampling
                  cmpi.b #'b',d7
                  beq enable.sampling

                  cmpi.b #'R',d7      read clock and send out
                  beq send.time
                  cmpi.b #'r',d7
                  beq send.time


*               Debugging code
*check.clock    bra entry.pnt

*               Check to see if time to sample
check.clock     bsr get.time            get time
                cmpi #0,d7              check for midnight
                bne skip.over
                move #$ffff,sys.go      enable sampling

*               Check for on tens of seconds
skip.over:      andi #$ff,d7            if min=0 then sample
                bne entry.pnt

*               If time for sample wait to prevent overrun
wait:           bsr get.time
```

```
                andi #$f,d7
                beq wait

*               Get time and call sampling routine
                bsr get.time
                bra sample

*               For debugging purposes
                bra entry.pnt


*---------------------------------------------------------------
*
*       tracer:
*
*       instruction trace calling routine

tracer:         trap #0         force execution of the trace
*                               exception



*---------------------------------------------------------------
*
*       enable.sampling:
*
*       Sets the flag SYS.GO to enable sampling

enable.sampling:
                move #$ffff,sys.go
                bra  entry.pnt
```

```
****************************************************************
*
*       ascii.hex :                             {subroutine}
*
*       This routine converts an ascii byte in d7 to a hex
*       nibble in d7. No other registers are altered. Illegal
*       data flagged with a return of $FF.
*
*
*       source file : asciihex.a68
*
*
*       Revisions:
*
*       ________________________________________________________
*          0.00          Norm Mortensen          28 Jan 87
*
*
****************************************************************

*       Entry point
        xdef    ascii.hex       * declare global variable for
                                * the linker
ascii.hex:
                movem.l a0-a1,-(sp)     * save the entry status
                lea     table(pc),a0    * point a0 to data table
                andi.l  #$ff,d7         * clear all but low byte
                andi    #$1e,ccr        * clear the carry
                subi    #$30,d7         * test for under range
                bcc     good.char       * and remove offset
                cmpi.b  #$17,d7         * check for over range
                bcs     good.char

bad.input:      move.l #$ff,d7          * return error symbol
                movem.l (sp)+,a0-a1     * return to entry state
                rts                     * go home!

        good.char:
                move.l d7,a1            * put offset into pointer
                move.b (a0,a1),d7       * get hex value
                andi.l  #$f,d7          *  save  only  low nybble
                movem.l (sp)+,a0-a1     * return to entry status
                rts                     * go home!


        table:                          * hex look up table
                dc.b 0,1,2,3,4,5,6,7,8,9
                dc.b $ff,$ff,$ff,$ff,$ff,$ff,$ff
                dc.b $a,$b,$c,$d,$e,$f,$ff
```

```
******************************************************************
*
*       clock.read                                   {subroutine}
*
*       Primative routine to read the contents of a register
*       of the real time clock. The number of the register to
*       be read is placed the low nibble of d7. The contents
*       of the register are returned in the low nibble of d7.
*       No other register contents are altered.
*
*
*       source file : clk_read.a68
*
*
*       revisions:
*
*       __________________________________________________
*         1.0   28Apr87         Norm Mortensen
*
*
******************************************************************

*       global definitions
        xdef clock.read

*       internal definitions
        a.status: equ $1f003
        a.data:   equ $1f001
        b.status: equ $1f007
        b.data:   equ $1f005

clock.read:

*       configure pia for read
        move.b #$30,a.status    get to direction reg
        move.b #$0f,a.data      0-3:outputs, 4-7:inputs
        move.b #$34,a.status    get back to the data reg

*       put address on lines
        move.b d7,a.data

*       run strobes for read cycle
        move.b #$e,b.data       activate clk select
        move.b #$c,b.data       activate *cs, *rd
        nop
        move.b a.data,d7        get data
        move.b #$e,b.data       only *cs active
        move.b #$f,b.data       all inactive, read done

*       position data
        asr #4,d7               slide data to low nibble
        andi #$f,d7             mask off the rest of the reg

*       make pia all inputs
        move.b #$30,a.status    get to direction reg
        move.b #0,a.data        all lines inputs
        move.b #$34,a.status    get back data reg

*       go back home
        rts
```

```
****************************************************************
*
*       clock.write                                 { subroutiine}
*
*       Primative routine to allow writing to the registers of
*       the real time clock. The data is in bits 7-4 of d7
*       the register number is in bits 3-0. No register contents
*       are changed by this routine.
*
*       source file : clk_wrt.a68
*
*       revisions:
*
*       ______________________________________________________
*         1.0   28Apr87          Norm Mortensen
*
*
****************************************************************

*       global definitions
        xdef clock.write
*       internal definitions
        a.status:       equ $1f003      use byte addressing mode
        a.data:         equ $1f001      only with these labels
        b.status:       equ $1f007
        b.data:         equ $1f005

clock.write:

*       Configure the pia for a clock write
        move.b #$30,a.status    get to data direction reg
        move.b #$ff,a.data      all bits outputs
        move.b #$34,a.status    back to data reg

*       Put data and address on line
        move.b d7,a.data

*       Go thru right sequence for the clock
        move.b #$e,b.data       activate clock select
        move.b #$a,b.data       activate select and wrt strobes
        nop                     give clock time
        move.b #$e,b.data       only clk select active
        move.b #$f,b.data       all inactive write complete

*       Make the pia all inputs and exit
        move.b #$30,a.status    get to data direction reg
        move.b #0,a.data        all bits inputs
        move.b #34,a.status     back to data reg
        rts
```

```
*****************************************************************
*
*       disp.reg                                      {subroutine}
*
*       This routine displays the contents of d7 on the the
*       terminal device. The output shows the register's
*       contents in hexadecimal form. No registers are
*       changed in this routine
*
*       source file : dispreg.a68
*
*       revisions:
*
*       _____________________________________________________
*         0.00          Norm Mortensen            23 Jan 87
*
*
*****************************************************************

*         define global label
          xdef  disp.reg

*         external references
          xref  hex.ascii        hex to ascii subroutine
          xref  putchar          display character

*         Internal Symbols
          CR:   equ     $0d
          LF:   equ     $0a

disp.reg:       movem.l d0-d2,-(sp)      save entry state of uP
                move.l  d7,d0            put data in d0
                move.l  #$f,d2           character mask
                move    #28,d1           init shift control reg.

nibble.out:     move.l  d0,d7            copy data
                asr.l   d1,d7            shift a nibble down
                and.l   d2,d7            clear off the extra bits
                bsr     hex.ascii        convert to ascii code
                bsr     putchar          put out serial port
                subi    #4,d1            dec. shift amount by 4
                bcc     nibble.out       if shift > 0 then cont.
                move.b  #CR,d7           close up the line with
                bsr     putchar          a carraige return and a
                move.b  #LF,d7           linefeed
                bsr     putchar
                move.l  d0,d7            put registers in order
                movem.l (sp)+,d0-d2      get back entry status
                rts
```

```
****************************************************************
*
*       disp.string:                                {subroutine}
*
*       disp.string displays a string on the output console
*       device. A pointer to the string to be displayed is
*       passed in register d7. The string should be stored in
*       increasing sequential memory positions. The end of
*       string is signaled with an ascii NULL. No registers
*       are changed.
*
*       source file : dispstr.a68
*
*       revisions:
*
*       _____________________________________________________
*          0.00         Norm Mortensen          30 Jan 87
*
*
****************************************************************


*       Entry point
        xdef    disp.string

*       External subroutine refences
        xref    putchar


disp.string:
                movem.l d0/d7/a0,-(sp)  * save the entry status
                move.l  d7,a0           * put string pointer in a0


output.chars:   move.b  (a0)+,d7
                beq     end.of.string   * if an ascii null quit
                bsr     putchar         * otherwise send out
                bra     output.chars    * go for next char


end.of.string:  movem.l (sp)+,d0/d7/a0  * return to entry state
                rts                     * go home!
```

```
*****************************************************************
*
*       download
*
*       this routine downloads files from the North Star to the
*       harmonic analyzer. The files must be in the form of
*       Motorola S records in order to be processed properly
*       by the harmonic analyzer.
*
*       source file :   download
*
*       language    :   Avocet 68000 assembler
*
*       revisions:
*
*       ______________________________________________________
*        1.00   6Mar87  Norman Mortensen
*
*
*****************************************************************

*       global label definitions
        xdef download

*       external references
        xref ascii.hex
        xref entry.pnt
        xref in

*       internal label definitions
ram.top         equ $27fffe

download:       bsr in
                cmpi.b #'S',d7          search until 'S' is found
                bne download
                bsr in                  get character after 'S'
                cmpi.b #'9',d7
                beq s9.line
                cmpi.b #'2',d7
                beq s2.line             then decode as s2 line
                cmpi.b #'1',d7          if s1 then fall thru
                bne download            try again

*
*       register assignmnets for S1 line decoding
*
*               d0 : counts number of data btes remaining in line
*               d1 : data and scratchpad register
*               d2 : address character counter
*               a0 : address pointer register
*
```

```
*               get the number of data bytes is a line

byte.count:     bsr in          get 1st count char
                bsr ascii.hex   convert to hex => d7
                asl #4,d7       make it the high digit of count
                move d7,d0      put partial cnt in d0
                bsr in          get 2nd char count
                bsr ascii.hex   convert to hex => d7
                add d7,d0       d0 : counts data bytes in line

*               initialize registers for address decoding
                move.l #0,d1
                move #4,d2

*               decode the next for bytes as the address

address.bytes:  bsr in
                bsr ascii.hex
                asl #4,d1
                add d7,d1
                subi #1,d2
                bne address.bytes
                addi.l #$20000,d1       loads program into RAM
                move.l d1,a0            a0 now holds the address
                subi #3,d0              adjust character counter


*               decode the remaining bytes in pairs as hex
*               data in ascii

data.bytes:     move #0,d1              init data byte storage
                bsr in                  get 1st char of pair
                bsr ascii.hex           convert to hex => d7
                move d7,d1              partial data => d1
                asl #4,d1               make it the high digit
                bsr in                  get 2nd char of the pair
                bsr ascii.hex           convert to hex
                add d7,d1               decoded data => d1
                move.b d1,(a0)+         store and inc addr ptr

*               check to see if still in ram
                cmpa.l #ram.top+2,a0
                bcs addr.good
                movea.l #ram.top,a0     if not pt to top ram
addr.good:      subi #1,d0              decrement data byte cntr
                bne data.bytes          cont till end of line
                bra download            if eol then go back to
*                                       line decoding
```

```
*
*   register assignments for the s2 line decoding
*
*                d0 : holds the number of data bytes in the line
*                d1 : data register and temp storage
*                d2 : address character counter
*                a0 : the address pointer
*


s2.line:         bsr in
                 bsr ascii.hex
                 andi #$f,d7
                 move d7,d0
                 asl #4,d0
                 bsr in
                 bsr ascii.hex
                 andi #$f,d7
                 add d7,d0          d0 holds the line byte count



                 move.l #0,d1
                 move #5,d2

s2.address:      bsr in
                 bsr ascii.hex
                 andi.l #$f,d7
                 asl.l #4,d1
                 add.l d7,d1
                 dbf d2,s2.address
                 move.l d1,a0
                 subi #4,d0               remove the counts for the
*                                         address and checksum bytes
*                                         from the byte count

                 cmpa.l #ram.top+2,a0
                 bcs s2.data
                 move.l #ram.top,a0


                 subi #1,d0               decrement the byte count
*                                         so that the looping
*                                         primative db can be used

s2.data:         bsr in
                 bsr ascii.hex
                 andi.l #$f,d7
                 move.l d7,d1
                 asl #4,d1
```

```
                    bsr in
                    bsr ascii.hex
                    andi #$f,d7
                    add d7,d1
                    move.b d1,(a0)+

                    cmpa.l #ram.top+2,a0
                    bcs s2.address.valid
                    move.l #ram.top,a0

s2.address.valid:
                    dbf d0,s2.data
                    bra download


s9.line:            bsr in
                    cmpi #$1a,d7
                    bne s9.line
                    bra entry.pnt
```

```
*****************************************************************
*
*       fft                                          {subroutine}
*
*       Routine to perform the fast fourier transform on
*       128 samples of data. The pointer to the beginning
*       of the data is passed in a6. The data is stored as
*       follows:
*               #0      real    (2 bytes)
*                       imag    (2 bytes)
*               #1      real    (2 bytes)
*                       imag    (2 bytes)
*                             :
*                             :
*               #127     ""          ""
*
*       The data format is complex rectangular fixed point.
*       The results are calculated in place. No registers
*       are altered by the routine.
*
*
*       source file : fft.a68
*
*
*       revisions:
*
*       _______________________________________________________
*        1.1    Norm Mortensen  Feb 87      debugged
*        1.2      "      "      May 87      fixed error in
*                                           the butterfly swaps
*
*****************************************************************

*       define global label
        xdef fft

*       external references
        xref sintbl

*       parameters that arte dependent on the length of the FFT

samples         equ     128             number of samples
iterations      equ     7               samples = 2**iterations
cos             equ     64              offset of cos in sin table
*                                       also cos = samples/2


*       parameters dependent upon the a/d

bits            equ     12              bits of a/d
offset          equ     2048            offset of the a/d
```

```
*       standard parameters

im              equ     2               offset to im part
c.next          equ     4               next complex number
rmask           reg     d0-d7/a0-a6     (all registers but sp)


*
*--------------------------------------------------------------
*


*               Save the initial machine status
fft:            movem.l rmask,-(sp)     save the entry state




*
*       This loop subtracts 2048 from all input values. This
*       converts the data from the a/d to 2's compliment form.
*       It also  zeroes the imaginary counterpart.
*

                move    #samples-1,d0   set the sample counter
                moveq   #0,d2           clr d2 serve as const 0
                move.l  a6,a0           a0 pnts to the data
        init:   move    (a0),d1         get data
                subi    #offset,d1      convert to 2's comp
                move    d1,(a0)+        put back in location
                move    d2,(a0)+        clr imag part of data
                dbf     d0,init         if not last sample then
*                                       branch to init

*
*       perform the number crunching of the FFT
*       based on three nested loops: the iteration loop,
*       the sequence loop and the pnt in sequence loop.
*
*       register assignments are as follows
*           a0 : seq counter
*           a1 : # of sequences reference
*           a2 : data pntr
*           a3 : pntr to the base of the sin table
*           a4 : cos offset in the sin table
*           a5 : half sequence offset
*           a6 : points to the start of data
*           d0 : iteration counter
*           d1 : pnt in sequence counter
*           d2 : number of pnts per sequence /2
```

```
*

*                initialize the registers
                moveq   #1,d0            iteration counter
                movea.l #1,a1            # of sequences
                move    #samples,d2
                asr     #1,d2            # pts per sequence / 2
                lea     sintbl(pc),a3    ptr to the sintable


*               do #iterations times
iter.loop:          move.l  a6,a2        pt to data
                    movea.l #0,a0        init seq counter

                    move.l #0,a5         make sure a5 is clr
                    move   d2,a5         copy pts.per.seq / 2
                    add    a5,a5
                    add    a5,a5         half.seq * 4 since data is
*                                        4 bytes. half seq offset


*                   do # sequences times
sequence.loop:          moveq   #0,d1   init pt in seq


*                       do pts in seq / 2 times

*                           calculate g(x) = m(x) + m(x+N/2)
*                           and store the results in m(x)
*                           calculate h(x) = m(x) - m(x+N/2)
*                           and store the results in m(x+N/2)

pt.seq.loop:                move (a2),d4        m(x).re => d4
                            move im(a2),d5      m(x).im => d5
                            move (a2,a5),d6     m(x+N/2).re =>d6
                            move im(a2,a5),d7   m(x+N/2).im =>d7
                            add  d4,d6          add re => d6
                            add  d5,d7          add im => d7


                            cmpi #4,d0          if iter. cntr less
                            bcs no.divide.1     then don't divide
                            asr  #1,d6          take care of 1/N
                            asr  #1,d7          a little each iter

no.divide.1:                move d6,(a2)        store at m(x).re
                            move d7,im(a2)      store at m(x).im
                            sub  (a2,a5),d4     (m(x)-m(x+N/2)).re
```

```
                             sub  im(a2,a5),d5    (m(x)-m(x+N/2)).im

                             cmpi #4,d0           if iter. cntr less
                             bcs no.divide.2      than 4 don't /2
                             asr  #1,d4           take care of 1/N
                             asr  #1,d5           a little each iter
no.divide.2:             move d4,(a2,a5)      store re to m(x+N/2)
                         move d5,im(a2,a5)    store im to m(x+N/2)

*                        multiple the resulting h(x) by Wn
*                        get the sin and cos values to calculate
*                        Wn. arg Wn = 2pi/N * (pt in seq) *
*                        2 ** iteration.cntr. The 2pi/N is already
*                        taken into account in the formation of
*                        the sin table.

*                        calculate the offset into sin/cos table
                         move d1,-(sp)        save pt in seq cntr
                         asl  d0,d1           pt.seq *2**iteration
                         move d1,a4           sin offset => a4
                         move (sp)+,d1        get pt in seq back

*                        get cos(Wn) values
                         move cos(a3,a4),d4
                         move d4,d6           Wn.re in d4,d6

*                        get -sin(Wn) values
                         move (a3,a4),d5      get sin
                         neg  d5              -sin
                         move d5,d7           Wn.im in d5,d7

*                        multiply h(x) * Wn => m(x+N/2)
                         muls (a2,a5),d4      m(x+N/2).re * Wn.re
                         muls im(a2,a5),d5    m(x+N/2).im * Wn.im
                         muls im(a2,a5),d6    m(x+N/2).im * Wn.re
                         muls (a2,a5),d7      m(x+N/2).re * Wn.im
                         sub.l d5,d4          result.re => d4
                         add.l d7,d6          result.im => d6
                         move  #bits-1,d5     set shift index
                         asr.l d5,d4          rescale result.re
                         asr.l d5,d6          rescale result.im
                         move  d4,(a2,a5)     result.re => m(x+N/2)
                         move  d6,im(a2,a5)   result.im => m(x+N/2)

*                        loop cntr and data pntr increment
                         addq  #1,d1          inc pt.in.seq
                         addq  #c.next,a2     next data element

*                    Test for end of sequence
                     cmp d1,d2        cmp pt contr with # pts
```

```
                    bne pt.seq.loop

*                 Test for the end of the iteration
                  adda a5,a2           pnt to head of nxt seqnce
                  addq #1,a0           increment the seq cntr
                  cmpa a0,a1           compare with # sequences
                  bne  sequence.loop   if last seq fall thru

*             Test for end of the fft calculations
              add  a1,a1               2 * # sequences
              asr  #1,d2               halve #pts.per.seq
              addq #1,d0               incthe iter cntr
              cmpi #iterations+1,d0    check for last iteration
              bne  iter.loop           it.cntr = last iter fall thru


*
*             Butterfly transfers
*

*             Register assignments for the butterfly transfers
*                 a0 : data ptr
*                 d0 : offset index to data element
*                 d1 : corresponding bit reverse ofset to data ele
*                 d2 : pt sequence counter
*                 d3 : bit swap counter
*                 d4 : copy of pt seq cntr to be bit swapped
*


*             initialization of parameters
              move.l a6,a0             pt at the data
              moveq  #0,d2             init cntr

*             do number of samples times
butterfly:        move d2,d4           copy cntr to bit swap
                  moveq #0,d1          clr reversed bits reg

*                 do number of bits in index times
                  move #iterations-1,d3  init bit swap counter
bit.swap:             roxr #1,d4       slide bit out 1 way
                      roxl #1,d1       and slide it in another
                  dbf d3,bit.swap      loop check

                  cmp d2,d1            d1<d2 if carry set. Don't
                  bcs swap.done        swap if d1<d2, already done
                      move.l d2,d0     put cntr in offset reg
                      asl.l #2,d0      *4 since data 4 bytes wide
                      asl.l #2,d1      *4 since data 4 bytes wide
                      move.l (a0,d0),d5        :
                        move.l (a0,d1),(a0,d0)  :  swap the data
                        move.l d5,(a0,d1)       :

swap.done:          addq #1,d2                 inc pnt cntr
                    cmpi #samples,d2           if d2=#samples then done
                    bne butterfly


*                   clean up after the fft
                    movem.l (sp)+,rmask          get back entry status
                    rts                          go home!
```

```
****************************************************************
*
*               a sin/cos lookup table must be linked in at
*               this point in the code
*
*               in order to change the length of the input
*               sequence to the fft the length of the sine
*               table must be changed.
*
****************************************************************

*
*        128 pnt FFT sine table
*

*        global label definition
         xdef sintbl

sintbl:

                dc.w      0,   100,   201,   300
                dc.w    399,   497,   594,   690
                dc.w    783,   875,   965,  1052
                dc.w   1137,  1219,  1299,  1375
                dc.w   1447,  1517,  1582,  1644
                dc.w   1702,  1756,  1805,  1850
                dc.w   1891,  1927,  1959,  1986
                dc.w   2008,  2025,  2037,  2045
                dc.w   2047,  2045,  2037,  2025
                dc.w   2008,  1986,  1959,  1927
                dc.w   1891,  1850,  1805,  1756
                dc.w   1702,  1644,  1582,  1517
                dc.w   1447,  1375,  1299,  1219
                dc.w   1137,  1052,   965,   875
                dc.w    783,   690,   594,   497
                dc.w    399,   300,   201,   100
                dc.w      0,  -100,  -201,  -300
                dc.w   -399,  -497,  -594,  -690
                dc.w   -783,  -875,  -965, -1052
                dc.w  -1137, -1219, -1299, -1375
                dc.w  -1447, -1517, -1582, -1644
                dc.w  -1702, -1756, -1805, -1850
                dc.w  -1891, -1927, -1959, -1986
                dc.w  -2008, -2025, -2037, -2045
```

```
*****************************************************************
*
*       getchar:                                      {subroutine}
*
*       gets a one byte character from the second serial port.
*       The character is returned in the low byte of d7.
*
*
*       source file : getchar.a68
*
*
*       revisions:
*
*       _____________________________________________________
*         0.00           Norm Mortensen            20 Jan 87
*
*
*****************************************************************


*       Entry point

        xdef    getchar


*       Internal definitions

stat.reg2       equu    $012001
data.reg2       equu    $012003



getchar:
                move.b  stat.reg2,d7    * get port status
                andi.b  #1,d7           * has a byte been received
                beq     getchar         * if not check again
                move.b  data.reg2,d7    * otherwise get character
                rts                     * go home!
```

```
*****************************************************************
*
*       get_data.a68                            {subroutine}
*
*       this routine drives the analog sampling port
*       voltage samples are taken for one cycle ( 128 ) times
*       and then current samples are taken for one cycle
*       this routine doesn't alter the contents of any register
*       Call as get.data
*
*       source file : get_data.a68
*
*       revisions:
*       ____________________________________________________
*        1.00   Norm Mortensen
*
*****************************************************************


*       define global labels
        xdef get.data

*       external references
        xref v.pntr
        xref i.pntr


*       register assignments
*
*               d0: sample counter
*               d1: data register
*               a0: data block pointer
*


*  constant declarations

a.control.reg   equu    $01f002
a.direction.reg equ     $01f000
a.data.reg      equ     $01f000

b.control.reg   equ     $01f006
b.direction.reg equ     $01f004
b.data.reg      equ     $01f004

entry.point     equ     $000458

threshold       equ     $50
samples         equ     128
```

```
*
*-------------------- Data Sampling Routine -------------------
*

get.data:
                movem.l d0/dl/d2/a0,-(sp)       save entry status
                move #0,d2                      clear d2

*               configure the a side of the pia
                move #$30,a.control.reg
                move #$00,a.direction.reg
                move #$34,a.control.reg

*               configure the a side of pia
                move #$00,b.control.reg
                move #$0f,b.direction.reg
                move #$04,b.control.reg
                move.b #$0f,b.data.reg

*               enable the a/d converter
                move #$07,b.data.reg            enable the a/d
                move.l #v.pntr,a0


*               Do two dummy conversions to set things up
con.1:          move.b a.control.reg+l,dl
                bpl con.1
con.2:          move.b a.control.reg+l,dl
                bpl con.2

*
*---------------- Zero crossing detector ------------------------
*

chk.threshold:  move.b a.control.reg+l,dl
                bpl chk.threshold               check for eoc

*               check to see if data below a certain threshold
                move.b a.data.reg+l,dl
                cmpi.b #threshold,dl
                bcc chk.threshold               if not, new sample

*               check for the - to + transition
zero.crossing:  move.b a.control.reg+l,dl
                bpl zero.crossing               wait until eoc
                move.b a.data.reg+l,dl          if transition then
                bpl zero.crossing               start sampling

*               store data
                move.b dl,(a0)+                 store msb
                move.b b.data.reg+l,(a0)+       store lsb
                move d2,(a0)+                   clear imag

*
*-------------------- voltage sampling --------------------
*

*               get the rest of the voltage samples
                move #samples-2,d0              set sample cntr
sample.v:       move.b a.control.reg+l,dl       check status
                bpl sample.v
                move.b a.data.reg+l,(a0)+       if eoc then grab
                move.b b.data.reg+l,(a0)+       result and store
                move d2,(a0)+                   clear imag
                dbf d0,sample.v
*               voltage samples done now

*
*-------------------- current sampling --------------------
*

*               set up for current sampes
                move.b #$3c,a.control.reg+l     switches mux
*                                               channel to i
                move.l #i.pntr,a0               pnt to start of
*                                               i block

*               get the current samples
                move #samples-l,d0              init sample cntr
sample.i:       move.b a.control.reg+l,dl       check status
                bpl sample.i
                move.b a.data.reg+l,(a0)+       if eoc grab 12 bit
                move.b b.data.reg+l,(a0)+       result and store
                move d2,(a0)+                   clear imag
                dbf  d0,sample.i
*               current samples done now

                move #$0f,b.data.reg        deselect the adc

*
*----------------------- unskew data -----------------------
*
*       sort out the samples and shift the data over so that
*       it's in proper format
*

                move #samples-l,d0      initialize sample counter
                move.l #v.pntr,a0       initialize the sample
*                                       pointer
```

```
v.sorter:       move (a0),d1            get skewed data word
                asr #4,d1               orient it properly
                andi #$0fff,d1          mask off the unused bits
                move d1,(a0)+           put it back
                lea 2(a0),a0            skip over the imag part
                dbf d0,v.sorter


                move #samples-1,d0      initialize sample counter
                move.l #i.pntr,a0       initialize the sample
*                                       pointer

i.sorter:       move (a0),d1            get skewed data
                asr #4,d1               orint properly
                andi #$0fff,d1          mask off unused bits
                move d1,(a0)+           put back, next
                lea 2(a0),a0            skip over imag part
                dbf d0,i.sorter

                movem.l (sp)+,d0/d1/d2/a0       recover original
*                                               data

                rts
*               jmp entry.point         alter to a rts
*                                       when tested
```

```
****************************************************************
*
*       get.time                                   {subroutine}
*
*       Gets the hours and minutes from the real time clock
*       and returns them in the low word of d7. The data is
*       in bcd form with tens of hours the most significant
*       nibble. No other registers are altered
*
*
*       source file : get_data.a68
*
*
*       revisions:
*       __________________________________________________
*         1.00  28Apr87  Norm Mortensen
*
*
****************************************************************

*       global definitions
        xdef get.time

*       external references
        xref clock.read

*       internal definitions
        sec.l:  equ     2       clock register numbers
        sec.10: equ     3       i.e. 10's of seconds ...
        min.l:  equ     4
        min.10: equ     5
        hrs.l:  equ     6
        hrs.10: equ     7


get.time:

*               Save machine status
                movem.l d0-dl,-(sp)


*               Read units of minutes
read.time       move #min.l,d7          Save the smallest unit
                bsr clock.read          of time read. If this
                move d7,d0              doesn't change other don't.


*               Clr dl for use as time storage reg
                moveq #0,dl
```

```
*                   Read in tens of hours
                    move #hrs.10,d7
                    bsr clock.read
                    add d7,dl
                    asl #4,dl

*                   Read in the units of hours
                    move #hrs.1,d7
                    bsr clock.read
                    add d7,dl
                    Asl #4,dl


*                   Read in tens of minutes
                    move #min.10,d7
                    bsr clock.read
                    add d7,dl
                    asl #4,dl


*                   Read in units of minutes
                    move #min.1,d7
                    bsr clock.read
                    add d7,dl

*                   Make sure time hasn't changed
                    cmp d0,d7
                    bne read.time


*                   If time hasn't changed then call it a clock read
                    move dl,d7
                    movem.l (sp)+,d0-dl
                    rts
```

```
****************************************************************
*
*       go                              {program fragment}
*
*       this routine begins program execution at a specified
*       memory location
*
*
*       source file :   go.a68
*       language:       Avocet 68000 assembler
*
*
*       revisions:
*
*       ______________________________________________________
*        1.00   6Mar87  Norman Mortensen
*
*
****************************************************************



*       global label definitions
        xdef go

*       external labels
        xref ascii.hex
        xref entry.pnt
        xref in

cr      equ     13

go:             move.l #0,d0            d0 addr pntr storage
ex.addr:        bsr in
                    cmpi #cr,d7
                    beq ex.addr.even    stop on cr
                    bsr ascii.hex
                    asl.l #4,d0
                    andi #$1e,ccr
                    add.l d7,d0
                bra ex.addr

ex.addr.even:   move d0,d7              only error protection
                asr #1,d7               against odd addresses
                bcs entry.pnt

                move.l d0,a0
                jmp (a0)
```

```
*****************************************************************
*
*       hex.ascii                                   {subroutine}
*
*       The hex.ascii routine converts a hex nibble in the
*       low nibble of d7 to an ascii character code. This
*       is returned in the low byte of d7. Since the low byte
*       of d7 is always hex no error status is returned. The
*       contents of no other registers are returned.
*
*
*
*       source file : hexascii.a68
*
*
*       Revisions:
*
*       _____________________________________________________
*          0.00         Norm Mortensen           28 Jan 87
*
*
*****************************************************************


*       Entry point
        xdef    hex.ascii                   * global variable for
                                            * the linker



hex.ascii:
                movem.l a0-a1,-(sp)         * save entry state of uP

                lea     table(pc),a0        * a0 points to the ascii table
                andi.l  #$f,d7              * clr all but the low nibble
                move.l  d7,a1               * put hex value in a pntr reg.
                move.b  (a0,a1),d7          * pull ascii eqv. from table
                andi.l  #$ff,d7             * clear all but the low byte

                movem.l (sp)+,a0-a1         * return reg to entry status
                rts                         * go home!



        table:                              * ascii code look up table
                dc.b '0123456789ABCDEF'
```

```
******************************************************************
*
*       hex.check                                   {subroutine}
*
*       Checks to see if the low byte of d7 countains an
*       ascii code that corresponds to a hexadecimal digit.
*       If the character is valid the carry bit is cleared
*       otherwise the character is set. No regisers are
*       changed.
*
*
*       source file : hexcheck.a68
*
*
*       revisions:
*
*       ______________________________________________________
*          0.00         Norm Mortensen          28 Jan 87
*
*
******************************************************************


*       Entry point

        xdef    hex.check


hex.check:
                cmpi.b  #'0',d7         * check to see if at
                bcs     return.bad      * least $30. If not bad
                cmpi.b  #'G',d7         * check to see if less
                bcc     return.bad      * than 'G'.If >= then bad
                cmpi.b  #'A',d7         * If what's left is >= A
                bcc     return.good     * then the code is good
                cmpi.b  #':',d7         * If what'S left is <= :
                bcs     return.good     * then it's good

return.bad      ori.b   #1,ccr          * clear the carry
                rts                     * go home!

return.good     andi.b  #$1e,ccr        * clear the carry
                rts                     * go home!
```

```
*****************************************************************
*
*       in                                          {subroutine}
*
*       this routine gets the next character input into acia 1
*
*
*       source file :   in.a68
*       language :      Avocet 68000 assembler
*
*
*       revisions:
*
*       _________________________________________________
*        1.00   6Mar87  Norman Mortensen
*
*
*****************************************************************

*       global label definition
        xdef in

*       external references
        xref putchar

*       label definitions
sregl:  equ $011000     use next higher odd addr if using
dregl:  equ $011002     byte addressing to these locations


in:             move sregl,d7           check for character
                andi #1,d7              until one is ready
                beq in

                move dregl,d7           received char => d7
                move d7,-(sp)           save on stack


echo.rdy:       move sregl,d7           check to see if xmitter
                andi #2,d7              is xmitting until it
                beq echo.rdy            isn't

                move (sp)+,d7           get char from stack
                andi.l #$7f,d7          clear all byte low 7 bits
                move d7,dregl           send as an echo
                bsr putchar             push out other serial prt
                rts                     go home!
```

```
****************************************************************
*
*       initialize                                  {subroutine}
*
*       Performs the initialization of the clock. The clock
*       is placed in the 24 hr mode. The hours and minutes
*       registers are then set. The time data is received via
*       the serial communication port. No register contents
*       are altered.
*
*
*       source file : init.a68
*
*
*       revisions:
*
*       ___________________________________________________
*         1.00  28Apr87  Norm Mortensen
*
*
****************************************************************

*       global definitions
        xdef initialize

*       external references
        xref ascii.hex
        xref clock.write
        xref data.pntr
        xref data.start
        xref entry.pnt
        xref in
        xref sample.cntr
        xref sys.go

*       internal definitions
        a.status: equ   $1f003
        a.data:   equ   $1f001
        b.status: equ   $1f007
        b.data:   equ   $1f005


initialize:

*               Save machine status and setup time
                movem.l d0-d3/d7,-(sp)

*               Get the time to set the clock to
                bsr in                  get tens hours
                move d7,d0
                bsr in                  get units of hours
                move d7,dl
                bsr in                  get tens of minutes
                move d7,d2
                bsr in                  get ubits of minutes
                move d7,d3


*               Initialize the system variables
                move.l #data.start,data.pntr
                move #0,sample.cntr
                move #0,sys.go


*               Initialize the clock
                move #5,d7
                bsr clock.write         stop clock, mask int's


*               Set to 24 hr mode
                move #$fl,d7
                bsr clock.write


*               Put in no interrupt mode
                move #7,d7
                bsr clock.write
                move #$f0,d7
                bsr clock.write

*               Set the tens of hours nibble
                move d0,d7
                bsr ascii.hex
                asl #4,d7
                addi #7,d7
                bsr clock.write

*               Set the units of hours nibble
                move dl,d7
                bsr ascii.hex
                asl #4,d7
                addi #6,d7
                bsr clock.write


*               Set the tens of minutes nibble
                move d2,d7
                bsr ascii.hex
                asl #4,d7
                addi #5,d7
                bsr clock.write
```

```
*                   Set the units of minutes nibble
                    move d3,d7
                    bsr ascii.hex
                    asl #4,d7
                    addi #4,d7
                    bsr clock.write


*                   Start up the clock
                    moveq #0,d7
                    bsr clock.write


*                   Get back machine status and back home
                    movem.l (sp)+,d0-d3/d7
                    jmp entry.pnt
```

```
*****************************************************************
*
*       memory
*
*       this routine allows the examination and alteration
*       of the harmonic analyzers memory by the North Star
*
*
*       source file :   memory.a68
*       language :      Avocet 68000 assembler
*
*
*       revisions:
*
*       ____________________________________________________
*        1.00   6Mar87  Norman Mortensen
*
*
*
*****************************************************************


*       global label definitions
        xdef memory


*       external references
        xref ascii.hex
        xref disp.reg
        xref entry.pnt
        xref hex.ascii
        xref in
        xref out
        xref putchar


*       internal label definitions
bs              equu $08        backspace
cr              equ $0d         carraige return
error           equ $00         error code
esc             equ $1b         escape
no.error        equ $01         a no error code
ram.top         equ $02fffe     top of the system ram
space           equ $20         space


*       Register assignments:
*         d0 : address pointer register
*         d1 : new data storage register
*         d2 : shift control register
*         d7 : i/o register
```

```
memory:         moveq #0,d0      clr the address storage reg
                moveq #0,d1      clr temp data storage reg


addr.fetch:
                bsr in                   get addr char
                cmpi.b #cr,d7            end of addr data?
                beq inrange              break out if end
                bsr ascii.hex            convert ascii to hex
                asl.l #4,d0              shift address up 1 digit
                add.l d7,d0              add in the low nibble
                bra addr.fetch           get next datum


inrange:        cmpi.l #ram.top+1,d0
                bcs even
                move #error,d7
                bsr out
                bra entry.pnt

even:           move d0,d7
                asr #1,d7
                bcc good.addr
                move #error,d7
                bsr out
                bra entry.pnt

good.addr:      move #no.error,d7
                bsr out



examine.alter:
                move #28,d2
send.addr:      move.l d0,d7
                asr.l d2,d7
                bsr hex.ascii
                bsr out
                subi #4,d2
                bcc send.addr

                move.l d0,a0
                move #12,d2
send.data:      move (a0),d7
                asr d2,d7
                bsr hex.ascii
                bsr out
                subi #4,d2
                bcc send.data

                move #4,d3
com.data:       bsr in
                cmpi #esc,d7
                beq entry.pnt
                cmpi #space,d7
                beq increment
                cmpi #bs,d7
                beq decrement

                bsr ascii.hex
                asl #4,d1
                add d7,d1
                subi #1,d3
                bne com.data
                move d1,(a0)


*               increment the addr ptr and check to see if
*               it still pnts at system memory

increment:      add.l #2,d0
                cmpi.l #ram.top+1,d0
                bcs examine.alter
                move.l #ram.top,d0
                bra examine.alter


*               decrement the addr ptr and check to see
*               that it still pts at system memory

decrement:      subi #2,d0
                bcc examine.alter
                move #0,d0
                bra examine.alter
```

```
*****************************************************************
*
*       out
*
*       Outputs data thru acia 1. The data is passed in the
*       low byte of d7. After the data is sent an echo must
*       received before the subroutine returns. No registers
*       altered by this routine.
*
*
*       source file :   out.a68
*       language :      Avocet 68000 assembler
*
*
*       revisions:
*       ___________________________________________________
*        1.00   6Mar87  Norm Mortensen
*
*
*****************************************************************

*       global label definitions
        xdef out

*       internal definitions
sregl           equ $011000
dregl           equ $011002

out:
                move d7,-(a7)
                tx.rdy:
                  move sregl,d7
                  andi #2,d7
                beq tx.rdy

                move (a7)+,d7
                move d7,dregl
                move d7,-(a7)

                echo.back:
                  move sregl,d7
                  andi #1,d7
                beq echo.back

                move dregl,d7
                move (a7)+,d7
                rts
```

```
*****************************************************************
*
*       putchar:                                  {subroutine}
*
*       puts a one byte character out the second serial port.
*       No register contents are changed. The value to be output
*       is passed in the low byte of d7.
*
*
*       source file : putchar.a68
*
*
*       revisions:
*
*       _______________________________________________________
*          0.00          Norm Mortensen           20 Jan 87
*
*
*****************************************************************


*       Entry point

        xdef    putchar


*       Internal definitions

stat.reg2       equ     $012000         these locations should
data.reg2       equ     $012002         be read as words



putchar:
                move    d7,-(sp)        save byte to send
    put.rdy:    move    stat.reg2,d7    get status
                andi    #2,d7           check to see if xmit rdy
                beq     put.rdy         if not check again
                move    (sp)+,d7        pull out data
                move    d7,data.reg2    send out data
                rts                     go home!
```

```
***********************************************************************
*
*       qk_fft.a68                                  {subroutine?}
*
*       Causes the harmonic analyzer to sample one cycle
*       of voltage and current data. The sampled data is then
*       processed by an FFT and the results are left in the
*       same spot as the data was. No registers are changed.
*       The call is made as qk.fft
*
*       source file : gk_fft.a68
*
*       revisions:
*       ____________________________________________________
*        1.00   Norm Mortensen
*
***********************************************************************

*       global label definitions
        xdef qk.fft
        xdef v.pntr
        xdef i.pntr

*       external references
        xref entry.pnt
        xref fft
        xref get.data
        xref out

*       internal definitions
v.pntr          equ $20800      pntr to voltage data block
i.pntr          equ $20A00      pntr to current data block
d.lgth          equ 1024        # of data bytes


*               save machine status
qk.fft:         movem.l d0/a6,-(sp)

*               get the data
                bsr get.data

*               Do an FFT on the voltage data
                move.l #v.pntr,a6
                bsr fft

*               Do an FFT on the current data
                move.l #i.pntr,a6
                bsr fft

*               send data out the serial port
```

```
                 move #d.lgth-1,d0       init cntr
                 move.l #v.pntr,a6       send off byte by byte
snd.fft.data:    move.b (a6)+,d7
                 bsr out
                 dbf d0,snd.fft.data

*                Go back to the monior
                 movem.l (sp)+,d0/a6
                 jmp entry.pnt
```

```
**************************************************************
*
*       sample                                   {programlet}
*
*       Samples the data, does an fft, stores the time
*       and desired data if the SYS.GO flag is set. The
*       time is passed in d7. No registers are altered.
*
*
*       source file : sample.a68
*
*
*       revisions:
*
*       ______________________________________________________
*         1.00  28Apr87   Norm Mortensen
*
*
**************************************************************

*       global definitions
        xdef sample

*       external references
        xref data.pntr
        xref entry.pnt
        xref fft
        xref get.data
        xref hex.ascii
        xref i.pntr
        xref sample.cntr
        xref sys.go
        xref v.pntr

sample:

*               Save machine status
                movem.l d0/d1/d7/a0/a1/a6,-(sp)

*               See if sys.go set all samples already taken
                move sys.go,d0          If sys.go = 0 then
                beq done                no sampling, return.
                cmpi #168,sample.cntr   If 168 samples already
                beq done                then no sample taken.

*               Get data and perform fft
                bsr get.data
                move.l #v.pntr,a6
                bsr fft                 fft on voltage samples
                move.l #i.pntr,a6
                bsr fft                 fft on current samples
```

```
*                 Convert the hrs digits of the time to an integer
                  asr #8,d7               put hours in low byte
                  andi #$ff,d7            and clear.
                  move d7,d0
                  asr #4,d0
                  moveq #10,d1            tens place weighting
                  mulu d1,d0
                  andi #$f,d7
                  add d0,d7               integer in d7 (word length)

                  move.l data.pntr,a0     a0 is data pointer
                  move d7,(a0)+           store time integer

*                 Store voltage data
                  move.l #v.pntr,a1       a1 pnts to data to store
                  moveq  #23,d0           0-23 harmonic to store
v.store.1         move.l (a1)+,(a0)+
                  dbf d0,v.store.1
                  addq.l #4,a1            skip 24th harmonic
                  moveq #19,d0            25-63 od harmonics
v.store.2         move.l (a1)+,(a0)+
                  addq.l #4,a1            skip even harmonics
                  dbf d0,v.store.2

*                 Store the current data
                  move.l #i.pntr,a1       a1 pnts to data to store
                  moveq #23,d0
i.store.1         move.l (a1)+,(a0)+
                  dbf d0,i.store.1
                  addq.l #4,a1            skip 24th harmonic
                  moveq #19,d0            25-63 odd harmonics
i.store.2         move.l (a1)+,(a0)+
                  addq.l #4,a1            skip even harmonics
                  dbf d0,i.store.2

                  addi #1,sample.cntr     inc sample counter
                  move.l a0,data.pntr     save data location

done              movem.l (sp)+,d0/d1/d7/a0/a1/a6
                  bra entry.pnt
```

```
***************************************************************
*
*       send.data                                   {programlet}
*
*       Sends data in one hour chuncks to the Z-171. The
*       sample groups taken are sent one group at a time. The
*       transmision format is
*
*        - send number of hours worth of data as an integer
*          in Intel order
*        - Send an integer that tells what hour the sample
*          was taken
*        - send 44 harmonics voltage (2 integers/harmonic)
*        - send 44 harmonics current (2 integers/harmonic)
*        - send checksum as integer in Intel order
*        - wait for reception of a character '*'
*        - start over with time integer
*
*       All integers are two bytes in length. No registers are
*       altered by this machine.
*
*
*       source file:    snd_data.a68
*
*       author:         Norm Mortensen
*
*       date:           28Apr87
*
***************************************************************

*       global definitions
        xdef send.data

*       external references
        xref data.start
        xref entry.pnt
        xref sample.cntr

*       internal references
        sregl:  equ $011001
        dregl:  equ $011003

send.data:
                movem.l d0-d2/d7/a0,-(sp)       save status

*               Send number of samples in Intel order
                move sample.cntr,d7
                bsr snd
                asr #8,d7
                bsr snd
```

```
*               Send the data packages if any
                moveq #0,d0            clear sample counter
                move.l #data.start,a0  a0 data pntr
chk.for.end:    cmp sample.cntr,d0     check for end data
                beq done

*               Send out time and data in Intel order
                moveq #0,d1            clr the checksum reg
                move #176,d2           Send 177 words of data
data.out:       move (a0)+,d7
                add d7,d1              accumulate checksum
                bsr snd
                asr #8,d7
                bsr snd
                dbf d2,data.out

*               Send the checksum in Intel order
                move d1,d7
                bsr snd                lsb of checksum
                asr #8,d7
                bsr snd                msb of checksum

*               Increment counter of samples sent and go again
                addq #1,d0
                bsr rcv                Wait for okay for next
                bra chk.for.end        data package.

*               Wrap up if done
done:           movem.l (sp)+,d0-d2/d7/a0
                bra entry.pnt

*================================================================
*
*       snd                             {local routine}
*
*       Outputs low byte of d7 out serial port 1. It does NOT
*       expect an echo. No registers are altered.
*
*================================================================

snd:            move d7,-(sp)          Save data on stack.
tx.rdy:         move.b sreg1,d7        Check to see if the
                andi #2,d7             transmit register is
                beq tx.rdy             empty.
                move (sp)+,d7          get data
                move.b d7,dreg1        send data
                rts
```

```
*================================================================
*
*       rcv                                {local routine}
*
*       Gets an input character from serial port one and
*       returns it in the low byte of d7. No other registers
*       are altered. It does NOT provide an echo.
*
*================================================================

rcv:            move.b sregl,d7         Check for reception.
                andi #1,d7              If no character continue
                beq rcv                 checking.
                move.b dregl,d7         get data
                rts
```

```
*****************************************************************
*
*       send.time                                    {programlet}
*
*       Sends the time of the harmonic analyzers real time
*       clock out the serial port in ascii coded digits.
*       The hours and minutes are sent (4 characters).
*       No registers are altered by this routine
*
*
*       source file:    snd_time.a68
*
*       author:         Norm Mortensen
*
*       date:           30Apr87
*
*****************************************************************

*       global definitions
        xdef send.time

*       external references
        xref clock.read
        xref entry.pnt
        xref get.time
        xref hex.ascii
        xref out


send.time:

*               Save machine status
                movem.l d0-d1/d7,-(sp)

*               Get time and send out
                bsr get.time
                move d7,d0
                move #12,d1             initialize shift reg
send.out        move d0,d7              get raw data
                asr d1,d7               shift to appropriate byte
                bsr hex.ascii           convet to ascii
                bsr out                 send
                subi #4,d1              adjust shift reg
                bcc send.out

*               Recover machine status an return
                movem.l (sp)+,d0-d1/d7
                bra entry.pnt
```

```
*****************************************************************
*
*       trace.a68                                {exception}
*
*       The following code is executed when a trace exception
*       occurs on the harmonic analyzer. The contents of all the
*       registers are displayed as well as the contents of the
*       pc and the ccr.
*
*
*       source file : trace.a68
*
*
*       Revisions:
*
*       _______________________________________________________
*          0.00          Norm Mortensen          20 Jan 87
*
*
*****************************************************************


*       global label definition
        xdef trace


*       external references
        xref    ascii.hex
        xref    getchar
        xref    hex.check
        xref    putchar
        xref    disp.reg
        xref    disp.string



*       Internal definitions

rmask1          reg     d0-d7/a0-a6
CR              equ     $0d
LF              equ     $0a
trace           equ     $1000

                org trace
                movem.l rmask1,-(sp)    * all register contents
                                        * saved on the stack
                move.l  sp,a0           * copy the stack pointer


                lea     newline(pc),a1
                move.l  a1,d7
                bsr     disp.string
                bsr     disp.string         * clear two blank lines


disp.d0:        lea.l   d0.str(pc),a1
                move.l  a1,d7               * point to the d0 header
                bsr     disp.string         * display the header
                move.l  (a0)+,d7            * copy d0 from the stack
                bsr     disp.reg

disp.d1:        lea.l   d1.str(pc),a1
                move.l  a1,d7               * display d1
                bsr     disp.string
                move.l  (a0)+,d7
                bsr     disp.reg

disp.d2:        lea.l   d2.str(pc),a1
                move.l  a1,d7               * display d2
                bsr     disp.string
                move.l  (a0)+,d7
                bsr     disp.reg

disp.d3:        lea.l   d3.str(pc),a1
                move.l  a1,d7               * display d3
                bsr     disp.string
                move.l  (a0)+,d7
                bsr     disp.reg

disp.d4:        lea.l   d4.str(pc),a1
                move.l  a1,d7               * display d4
                bsr     disp.string
                move.l  (a0)+,d7
                bsr     disp.reg


disp.d5:        lea.l   d5.str(pc),a1
                move.l  a1,d7               * display d5
                bsr     disp.string
                move.l  (a0)+,d7
                bsr     disp.reg

disp.d6:        lea.l   d6.str(pc),a1
```

```
                move.l   al,d7           * display d6
                bsr      disp.string
                move.l   (a0)+,d7
                bsr      disp.reg


disp.d7:        lea.l    d7.str(pc),al
                move.l   al,d7           * display d7
                bsr      disp.string
                move.l   (a0)+,d7
                bsr      disp.reg


disp.a0:        lea.l    a0.str(pc),al
                move.l   al,d7           * display a0
                bsr      disp.string
                move.l   (a0)+,d7
                bsr      disp.reg


disp.al:        lea.l    al.str(pc),al
                move.l   al,d7           * display al
                bsr      disp.string
                move.l   (a0)+,d7
                bsr      disp.reg


disp.a2:        lea.l    a2.str(pc),al
                move.l   al,d7           * display a2
                bsr      disp.string
                move.l   (a0)+,d7
                bsr      disp.reg


disp.a3:        lea.l    a3.str(pc),al
                move.l   al,d7           * display a3
                bsr      disp.string
                move.l   (a0)+,d7
                bsr      disp.reg


disp.a4:        lea.l    a4.str(pc),al
                move.l   al,d7           * display a4
                bsr      disp.string
                move.l   (a0)+,d7
                bsr      disp.reg


disp.a5:        lea.l    a5.str(pc),al
                move.l   al,d7           * display a5
                bsr     disp.string
                move.l  (a0)+,d7
                bsr     disp.reg


disp.a6:        lea.l   a6.str(pc),al
                move.l  al,d7           display a6
                bsr     disp.string
                move.l  (a0)+,d7
                bsr     disp.reg


disp.a7:        lea.l   a7.str(pc),al
                move.l  al,d7           a7 is the stack pointer
                bsr     disp.string     In order to compensate for
                move.l  a0,d7           the use of the SP
                andi.b  #$le,ccr        in processing the trace
                addi.l  #6,d7           exception an offset
                bsr     disp.reg        is added


disp.sr:        lea.l   sr.str(pc),al
                move.l  al,d7
                bsr     disp.string
                move.l  #0,d7           clears the reg since the
                move    (a0)+,d7        SR is only a word wide
                bsr     disp.reg


disp.pc:        lea.l   pc.str(pc),al
                move.l  al,d7           program counter the next
                bsr     disp.string     long word on the stack
                move.l  (a0),d7         don't increment a0 so the
                bsr     disp.reg        sr can be reached with a
                                        predecremented read


*       Toggle the t bit of the status register


                move    -(a0),d7        get the sr
                bmi     trace.on
trace.off:      lea.l   t0.str(pc),al
                move.l  al,d7           display trace off message
                bsr     disp.string
                bra     cont.l

trace.on:       lea.l   tl.str(pc),al
```

```
                move.l  al,d7              display trace on message
                bsr     disp.string


cont.l:         lea.l   t.str(pc),al
                move.l  al,d7              display direction for
                bsr     disp.string        trace

                bsr     getchar            get answer
                cmpi.b  #CR,d7             if CR then don't change
                beq     no.toggle.t        the status of t

toggle.t:       eori    #$8000,(a0)        toggle the t bit
no.toggle.t:    move    (a0)+,d7           point back at the pc on
                                           the stack

                lea     newline(pc),al
                move.l  al,d7
                bsr     disp.string        go to a new line



*       Display the header for program counter alterations and
*       see if that is to be changed.


                lea.l   pc.str(pc),al
                move.l  al,d7
                bsr     disp.string

                move.l  #0,d0              clear pc storage reg


                bsr     getchar
                cmpi.b  #CR,d7
                beq     return.trace       if CR then exit trace

                bsr     hex.check          C=0 if the byte in d7 is
                bcs     next.char          hex digit in ascii. If bad
                                           wait for a new input
                bsr     putchar            echo to the screen
                bsr     ascii.hex          convert to hex
                andi    #$le,ccr           clear the carry
                add.l   d7,d0              put in low nible of
*                                          pc storage register
```

```
next.char:      bsr     getchar
                cmpi.b  #CR,d7
                bne     cont.2              If a CR put d0 into the
                move.l  d0,(a0)             pc's location on the stack
                bra     return.trace        and return from the trace

cont.2:         bsr     hex.check           C=0 means good character
                bcs     next.char           no echo if bad char
                bsr     putchar             echo
                bsr     ascii.hex           convert to hex
                asl.l   #4,d0               shift by one hex digit
                andi    #$1e,ccr            clear the carry
                add.l   d7,d0
                bra     next.char

return.trace:   lea     newline(pc),a1
                move.l  a1,d7
                bsr     disp.string
                bsr     disp.string         clear two blank lines

                movem.l (sp)+,rmask1        get back entry state
                rte                         go back
```

```
*****************************************************************
*
*       messages                          {constant definitions}
*
*       this file countains the messages used by the trace
*       exception. It is included as part of the trace file
*
*
*       revisions:
*
*      ____________________________________________________
*           0.00        Norm Mortensen           3 Feb 87
*
*
*****************************************************************

a0.str          db 'a0: ',0
a1.str          db 'a1: ',0
a2.str          db 'a2: ',0
a3.str          db 'a3: ',0
a4.str          db 'a4: ',0
a5.str          db 'a5: ',0
a6.str          db 'a6: ',0
a7.str          db 'a7: ',0

d0.str          db 'd0: ',0
d1.str          db 'd1: ',0
d2.str          db 'd2: ',0
d3.str          db 'd3: ',0
d4.str          db 'd4: ',0
d5.str          db 'd5: ',0
d6.str          db 'd6: ',0
d7.str          db 'd7: ',0

pc.str          db 'pc: ',0
sr.str          db 'sr: ',0

t0.str          db 'trace is disabled',CR,LF,0
t1.str          db 'trace is enabled',CR,LF,0

t.str           db 'trace: ',0

alter.pc        db 'to keep the pc the same '
                db 'type <return>',CR,LF,0
                db 'to alter type in the new '
                db 'address',CR,LF,0

newline         db CR,LF,0
```

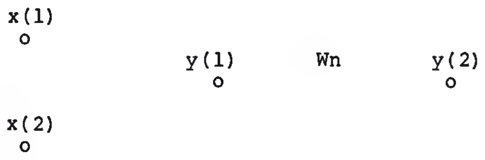
APPENDIX C

Source code listings of SETUP, the Zenith Z-171 interface program. For instructions on the operation of SETUP see [7].

APPENDIX D : SIGNAL FLOW GRAPHS

Signal flow graphs are a graphical way of representing the linear relationships between a set of input variables and a set of output variables. Signal flow graphs are useful in a number of areas. These include graphic representation of signal processing algorithms, and linear systems. This appendix describes how a signal flow graph is read.

In order to explain how to read a signal flow graph the simple figure below will be used.

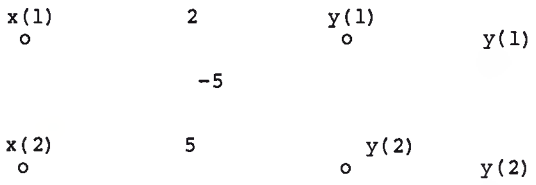


The variables x(1), x(2), y(1) and y(2) are represented by the nodes. Whenever two signals (lines) meet at a node their values are added to give the value of the node (variable). In the figure above y(1) = x(1) + x(2). The value of a signal (line) is the value of the node it originated from. This is then multiplied by any scaling factor indicated beside the line. If no scaling factor is indicated then it is assumed to be 1. In the example above the signal flowing from x(1) to y(1) has the value x(1). The signal flowing from y(1) to y(2) has the value y(1)Wn. Since this is the only signal flowing into the y(2) node y(2) = y(1)Wn.

As another example the signal flow graph for the following equations is shown below.

$$y(1) = 2x(1) - 5x(2)$$

$$y(2) = 5x(2) + y(1)$$



REFERENCES

1. Motorola, 16-Bit Microprocessor User's Manual 3ed., Prentice-Hall, Englewood Cliffs, N.J., 1982.

2. J.D. Greenfield and W.C. Wray, Using Microprocessors and Microcomputers the 6800 Family, Wiley and Sons, New York, N.Y., 1981.

3. N. Ahmed and T. Natarajan, Dicrete Time Signals and Systems, Reston, Reston, V.A., 1983.

4. W.D. Stanley, Digital Signal Processing, Reston, Reston, V.A., 1975.

5. N.K. Bose, Digital Filters : Theory and Applications, North-Holland, New York, N.Y., 1985.

6. P.D. Welch, "The use of Fast Fourier Transform for the Estimation of Power Spectra : A Method Based on Time Averaging Over Short Modified Periodigrams," IEEE Trans. Audio and Electroacoust., Vol. AU-15, pp. 70-73, June 1967.

7. G.L. Johnson, A.N. Mortensen, A.J. Heber, Measurement of Harmonics on Kansas Utilities, Report to Kansas Electric Utilities Research Program (KEURP), September, 1987.

8. Analog Devices, Data Aquisition Handbook, Analog Devices, 1985.

AN AUTOMATED HARMONIC ANALYZER FOR USE IN STUDYING THE COMMERCIAL POWER SYSTEM

by

NORMAN MORTENSEN

BSEE, Kansas State University, 1986

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Abstract

This document discusses the design of an automated harmonic analyzer for use in studying the commercial power system. This unit was developed in order to complete a study of harmonics on Kansas utilities sponsored by the Kansas Electric Utiities Research Program (KEURP). A brief introduction of the causes and possible effects of power system harmonics is presented. The physical design and operation of the harmonic analyzer are examined. The software for the harmonic analyzer's control program, ARDVARK, is explained on the flow chart level. The testing procedures used to evaluate the harmonic analyzer are explained and the results of these tests are presented.

The harmonic analyzer presented in this document has the following capabilities and features. It is a portable unit capable of independent operation for one week when properly initialized. It measures the first sixty four harmonics of 60 Hz. The analyzer's measurement error was found to be 0.1% of the magnitude of the 60 Hz fundamental waveform based on the evaluation tests described in this document.