MORSE CODE DETECTION AND TRANSLATION IN REAL TIME

by

Jeff McDow

Prepared for

School of Engineering

Walla Walla College

Submitted in Partial Fulfillment of

ENGR 496, 497, 498

May 30, 1991

ABSTRACT

Two systems capable of detecting and translating Morse code in real time were constructed. One system uses a phase-lock loop to detect the Morse code signal and the other uses an AT&T Digital Signal Processing (DSP) board. A computer program written in C translates the signal that has been detected by either of these methods. The phase-lock loop based system is capable of accurately translating Morse code signals up to 40 words per minute. The DSP-based system can errorlessly translate signals of up to 60 words per minute and can also simultaneously translate two superimposed signals. Currently, the DSP system is only able to translate relatively noise-free Morse code.

CONTENTS

CONTENTS (Continued)

TABLE OF FIGURES

MORSE CODE DETECTION AND TRANSLATION IN REAL TIME

INTRODUCTION

Background. Morse code is a means of electrical communication that was developed by Samuel Morse in the early 1800's for use with the telegraph. Despite technology that allows voice communication to any part of the world, Morse code is still being used today--primarily by amateur radio operators.

An alphanumeric character is represented in Morse code by a sequence of elements called dits and dahs. These elements are generated by keying a sine wave on and off for specific periods of time. For a dah, the sine wave is keyed three times longer than for a dit. By varying the length of the elements and the spacing between them, signals of different speeds can be sent.

Most amateur radio operators translate Morse code themselves. However, with the advent of the computer it is now possible to translate Morse code entirely by machine. This allows a person to understand Morse code messages that are being sent faster than he or she can translate.

Scope of Project. Work performed previously at Walla Walla College by H.R. Frohne and K. Jonsson had dealt with ways of detecting a Morse code signal. The goal of my project was to

expand upon this previous work by constructing a Morse code translation system consisting of a detection unit, a computer interface, and a computer program to translate the Morse code signal and display the result.

DETECTION HARDWARE

Because Morse code signals are sent asynchronously and consist of elements--dits and dahs--of unequal lengths, effective Morse code detection requires hardware that can immediately sense a Morse code element and time its duration. A simple way to detect Morse code would be to use a peak detector which would turn on when the magnitude of its input signal exceeded a certain threshold--such as when the Morse code signal was on. However, such a detector would not be very accurate if there was a high level of background noise, or if signals other than the desired one were present.

Two designs that have advantages over the simple peak detector utilize the phase-lock loop and the Fourier transform respectively [1, 2]. Both of these designs were implemented for this project.

Phase-Lock Loop. A phase-lock loop is a device which utilizes feedback to lock onto the frequency of an input signal. The LM567 tone decoder contains a phase-lock loop and has a pin

that is set at a low logic level whenever the phase-lock loop has achieved a lock. When the lock-in frequency of the phase-lock loop is adjusted to the same frequency as the Morse code signal, the LM567 will lock onto the signal and set the lock-indicator pin (pin 8) low. By timing how long this pin is at a low logic level, one can determine the duration of a Morse code element.

The phase-lock loop circuit used for this project is shown in Figure 1, and is based on the design of H.R. Frohne [1]. By adjusting the value of R_1, the lock-in frequency of the LM567 is adjusted according to $f \propto 1/RC$. To detect a Morse code signal, the signal is attached as shown, and the potentiometer, R_1, is adjusted until the LED flashes in unison with the sound of the signal. The digital signal from pin 8, indicating the presence and duration of dits and dahs, is sent to a computer for processing.

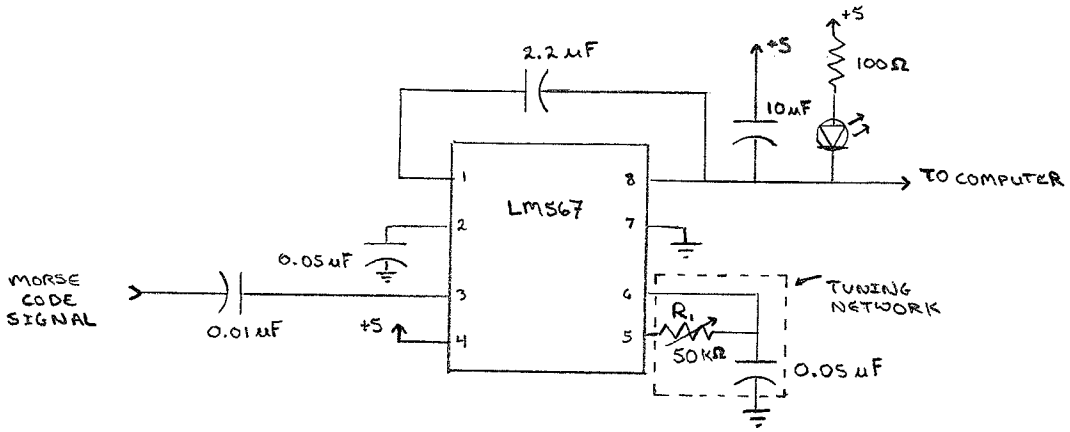


Figure 1: A Morse Code Detector Using a Phase-Lock Loop [1]

Fast Fourier Transform. A more complex, but more powerful way to detect a Morse code signal is to perform Fast Fourier Transforms (FFTs) on it at closely spaced intervals of time. If a Morse code signal is present, the FFT will show a strong magnitude component at the frequency of the signal. Conversely, if no signal is present the FFT will not have a well-defined peak at this frequency. The power of this method of detection is that in addition to effectively reducing background noise caused by other signals, it does not need to be manually adjusted in order to detect signals at different frequencies, and it is capable of accurately receiving two or more signals at once [2].

Figures 2 and 3 illustrate how the FFT can separate two superimposed Morse code signals into separate frequency components. Figure 2 shows the superposition of two Morse code signals of different frequencies. By performing a FFT on this signal, the magnitude of the component due to each of the frequencies is clearly seen, as shown in Figure 3a. Similarly, if only one of the signals is present the FFT will appear as shown in Figure 3b. Thus both signals can be effectively sampled by performing FFTs on the composite signal.

In order to perform Fourier transforms at a fast enough rate to allow real-time processing of a Morse code signal, the AT&T digital signal processing (DSP) board was used. Previously, Kristjan Jonsson had written software for the DSP board which could perform a FFT on a real-time signal [2]. The work performed for the current project expanded and modified these

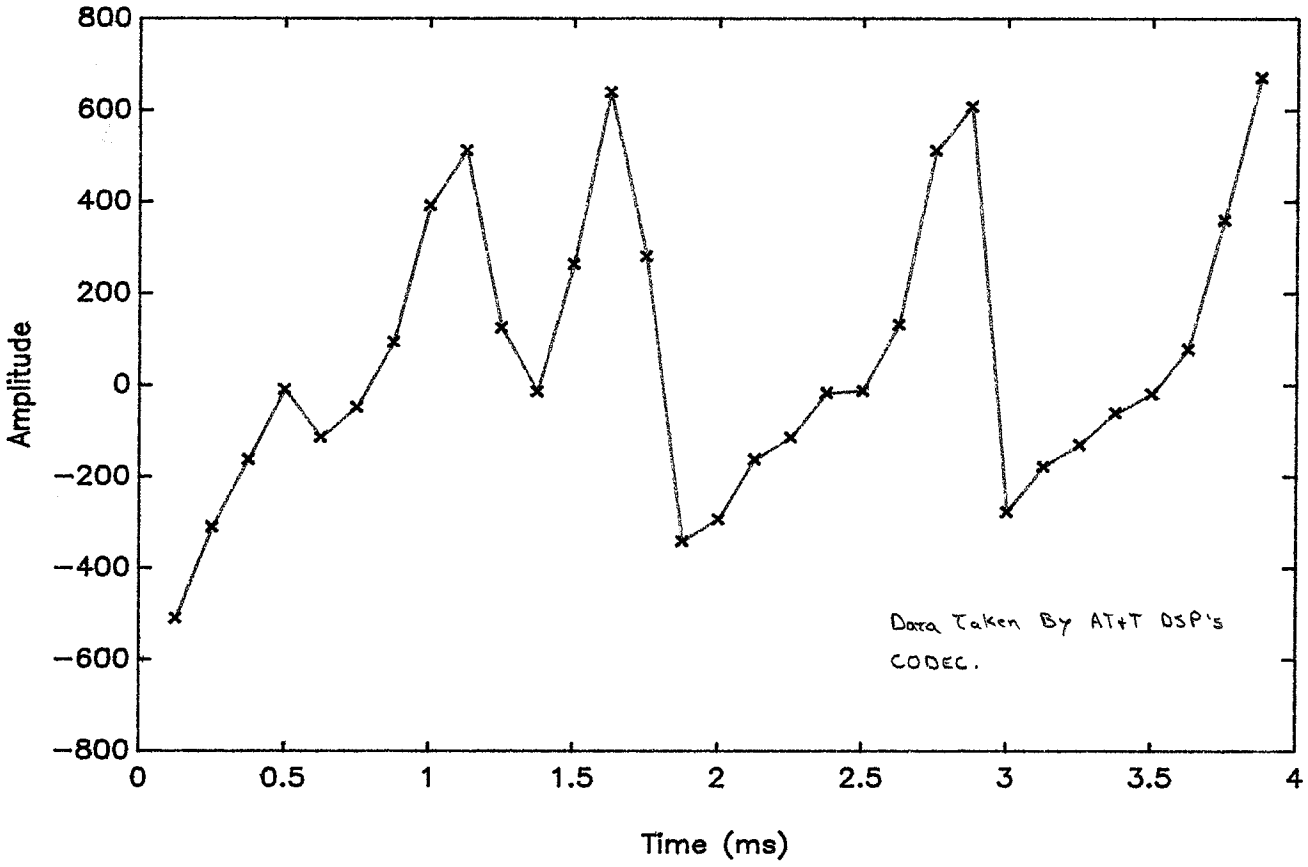


Figure 2: Two Superimposed Morse Code Signals

programs, making it possible to perform FFTs on a Morse code signal, interpret the results of these FFTs, and output a translation of the Morse code--all in real time.

COMPUTER INTERFACE

In order to interpret a Morse code signal that has been detected by either the phase-lock loop or DSP board, it is necessary to interface the detection hardware to a computer. While the AT&T DSP has its computer interface built in, there are a variety of ways to provide the interface for the phase-lock loop. Although the phase-lock loop only requires a single,

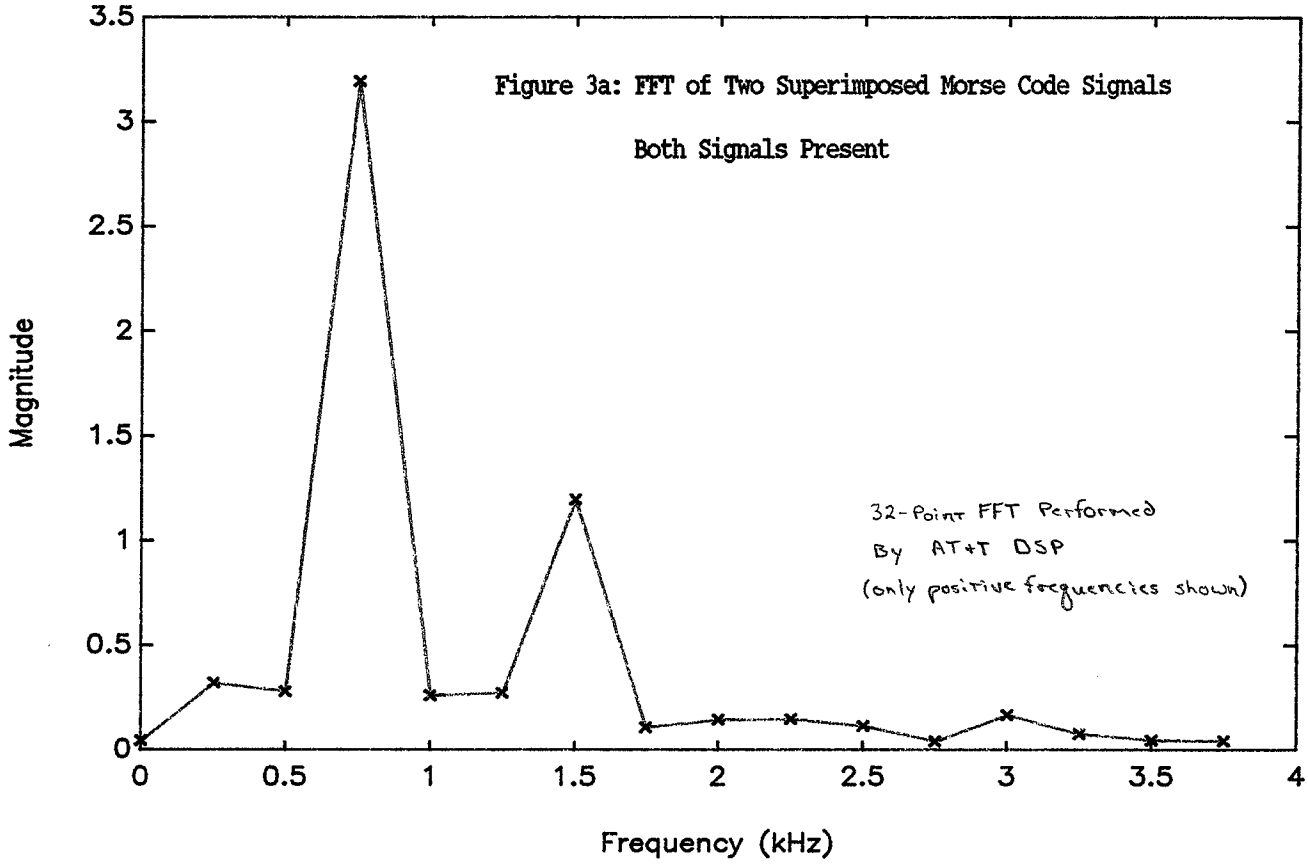


Figure 3a: FFT of Two Superimposed Morse Code Signals
Both Signals Present
32-Point FFT Performed
By AT+T DSP
(only positive frequencies shown)
Magnitude
Frequency (kHz)
0
0.5
1
1.5
2
2.5
3
3.5
4

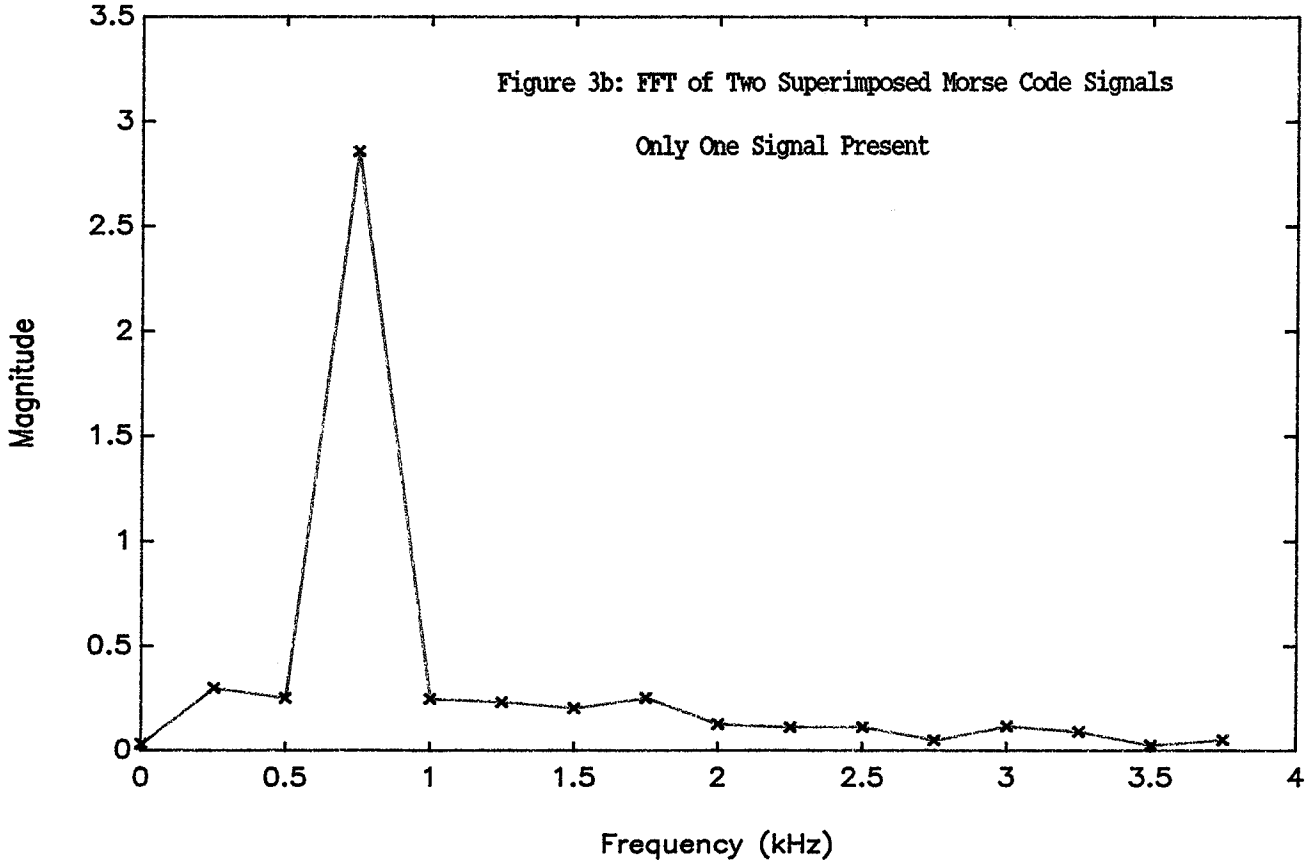


Figure 3b: FFT of Two Superimposed Morse Code Signals
Only One Signal Present
Magnitude
Frequency (kHz)
0
0.5
1
1.5
2
2.5
3
3.5
4

read-only interface line that can easily be supplied by a computer's game port or serial port, a custom-made interface card was instead chosen to provide the computer interface. This choice was made because the computer on which the Morse code translation program was to run did not contain a game port. Also, I wanted to have an expandable interface which would allow the connection of multiple tone decoders to be used in other applications such as touch-tone telephone decoding.

Interface Card. The fundamental component of the computer interface for the phase-lock loop tone detector is the JDR-PR2 interface card. This card fits into one of the expansion slots of an IBM-compatible personal computer and, once certain decoding circuitry is connected, can provide read and write capabilities to external hardware [3]. In addition to the circuitry suggested by the manufacturers of the card, I found it necessary to connect the hardware shown in Figure 4 (see Appendix A for complete schematics) [4].

Input from the tone decoder(s) is buffered by the LS541 tri-state buffer. When a read operation is performed to the correct address--in this case 300 hexadecimal--the buffer is enabled and data from the tone decoder(s) drive the data bus. Otherwise, the buffer is in a high impedance state. Similarly, when a write operation is performed, (this feature was not used in this project), the rising edge of the write signal, as well as a signal from the LS155 indicating that the write operation is

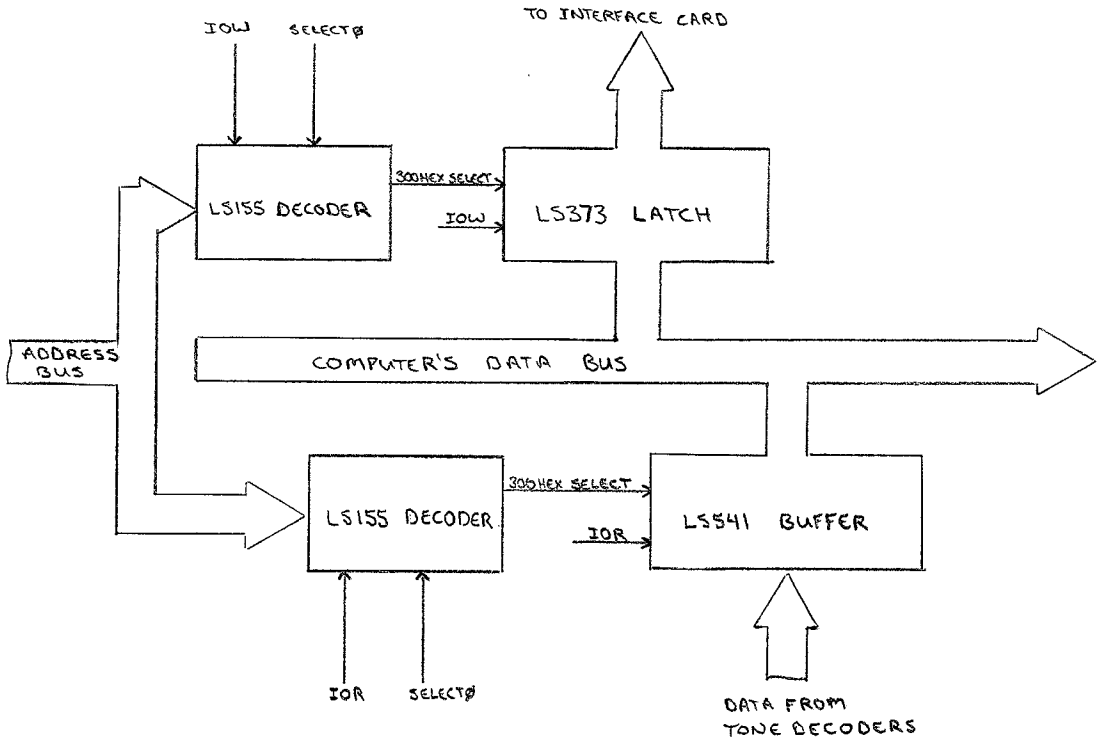


Figure 4: Interface Card Circuitry

directed to address 300 hex, cause the LS373 to latch the data off the computer's data bus so that it will not be lost when the write cycle is over.

DSP Board. The interface to the DSP board is a CODEC A/D converter which samples the incoming signal at 8 kHz. This data is input directly to the DSP which operates independently of the microprocessor of the computer to which the DSP board is connected [8].

Because the DSP is performing its processing at the same time as the main computer is running the Morse code translation

program, a special means of communication, called handshaking, is needed to let one processor know when the other processor needs or wants data.

The handshaking process is facilitated by monitoring the value of a variable common to the two processors. When the main processor needs the DSP to perform a FFT on the Morse code signal, it sets this variable equal to one. When the variable is set to one, the DSP performs the FFT and sets the variable back to zero. Setting the common variable to zero signals the translation program to load the contents of the DSP variables containing the FFT information. When the main computer desires another FFT to be performed, the handshake variable is set to one and the process is repeated.

TRANSLATION PROGRAM

The purpose of the translation program is to process the information that the detection hardware has sent to the computer via the interface. In both the phase-lock loop and DSP board designs, the information that is input by the detection hardware consists of a series of binary values (one for each time a read operation is performed) indicating whether a Morse code signal is present or not. The translation program processes these samples and outputs the translation to the computer's monitor. Complete listings of the translation programs used by the phase-lock loop system and the DSP board are found in Appendix D.

Flow of Operation. A simplified flowchart illustrating how the translation program works is shown in Figure 5 (for more detailed flowcharts see Appendix D). First, the program times the duration of various code elements, searching for the longest element which it calls a dah. Note that the timing is not performed by a clock, but instead is accomplished by counting the

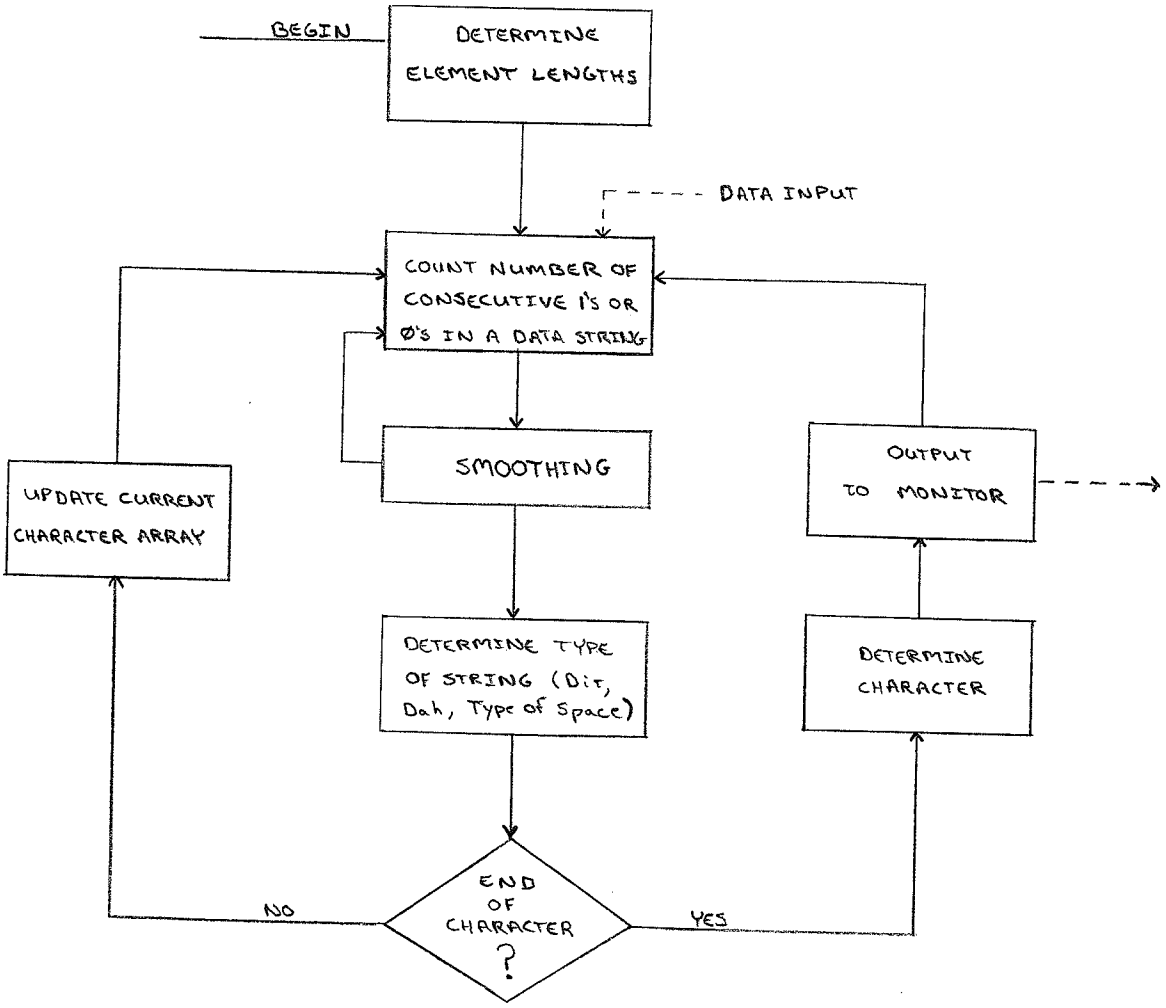


Figure 5: Translation Program Flowchart

number of consecutive samples of the same binary value that are received from the interface. From the number of samples in a dah, determined above, the program calculates the number of samples in a dit, an intercharacter space, an intracharacter space, and a word space using the standards given by the Amateur Radio Relay League [5].

Once the program has determined this rudimentary information, it begins translating the Morse code signal. This is done by counting the number of consecutive samples that the signal is present or absent and determining when a dit, a dah, or any of the various spacings has been received. After the program has determined the element's identity, it stores a 1 for each dit and a 2 for each dah in a special array used to keep track of the character that is currently being received. When the program detects an intercharacter space or a word space, the array of 1's and 2's is converted into a base 10 number by treating the array as a base 3 number. Next, the base 10 number is entered into a look-up table containing all the different Morse code characters. When a match is found in this table, the corresponding character is printed on the monitor and the character array is set to zero. (see [6] for a compilation of Morse code characters).

As an example, consider the processing of the letter "n", which consists of a dah followed by a dit. From the samples that it receives from the detection hardware, the program determines that a dah has been sent. The program will also determine that the dah is part of a character (not a character itself) because

an intracharacter space follows it. Therefore, a value of 2 is stored in the array used to keep track of the current character. Similarly, when the dit is received a 1 will be stored in the character array. When an intercharacter space or a word space is received following the "n", the program will assign the character array its base 10 equivalent value as shown below:

$$2 * 3^{1} + 1 * 3^{0} = 7$$

Entering the look-up table with this value, it is found that the number 7 corresponds to the character "n". Therefore, an "n" is printed on the computer's screen.

Smoothing and Sampling. Due to imperfections in the operation of the detection hardware, the input to the interface program may not consist of a simple string of binary values corresponding to whether the signal is on or off. At times a zero (on) will be received during a space (a string of 1's) or a one will be received when the signal is on (a string of 0's). Elimination of this problem relies on the fact that strings of erroneous samples are short in duration compared to the actual signal strings and can be corrected by smoothing the input data. This smoothing is done by simply counting the number of consecutive ones or zeros and comparing this count to some minimum value (determined during the initial setup time and modified during processing) corresponding to the presence of at least a dit (or, equivalently, an intracharacter space). If the count is less than this minimum, it is stored temporarily and a

new count is started. The new count is similarly processed and if it too does not exceed the threshold value for a dit, it is added to the previously stored count value. This process is repeated until a signal that is at least as long as a dit is received. At this time the sum of the undetermined counts (and whether they were predominately ones or zeros) is looked at to see whether they constitute a space, an element, or whether they are a glitch in the signal. If they constitute a glitch, their total is added to the count value of an adjacent dit, dah, or space depending on the level (on or off) of the majority of samples in the undetermined string.

As an example, consider the binary number string shown in Figure 6 in which 1's represent the signal being off and 0's represent the signal being on. Also assume that the program has previously determined that a dah has a minimum duration of 7 samples and a dit has a minimum duration of 2 samples.

```
Before Smoothing:
          1 1 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 1 1 1 0 ...

After Smoothing:
          1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 ...
```

Figure 6: Binary Number String From Morse Code Detector

When the first string of three 1's shown in Figure 6 is received, it is tentatively labeled as an intracharacter space. However, the single zero that follows does not constitute a string of at least two samples, so a count of one is stored

temporarily. When the succeeding string of four 1's is received, the value of the temporary count is reviewed, and since its length (one) is less than that of a dit, it is changed to a 1 and added to the string of three 1's preceding it; which, coupled with the succeeding string of four 1's, forms a string of eight 1's--an intercharacter space. In a similar fashion, the single 1 in the string of 0's is smoothed into a 0. Therefore the final interpretation of the binary string in Figure 6 is an intercharacter space followed by a dah, followed by an intracharacter space.

User Interface. The user interface for the translation program is menu-driven and relatively simple to use. When the program is started, it indicates that it is determining the duration of the dits and dahs. When the user presses "return", the program starts to translate the Morse code signal and output the translation to the screen. If the translation appears inaccurate, the user can either manually adjust the parameters that the program has determined, or have the program redetermine the parameters itself. Pressing "q" terminates program execution and returns the user to DOS. A complete guide to the code translation program is found in Appendix C.

TESTS

The following section describes tests that were run using the detection hardware and software discussed above.

Specifically, it was desired to see how fast each method of detection, operating with its corresponding software, could receive and translate Morse code with a high degree of accuracy. It was also desired to see if the DSP-based translation system could receive and translate two or more signals simultaneously. In addition, the effect of background noise on the ability of the DSP system to translate a Morse code signal was explored. Finally, a test was performed to see if the translation program could track a signal whose speed varied.

Speed Test: Phase-Lock Loop. The maximum speed that the phase-lock loop design was able to translate Morse code was determined by taping Morse code signals of various speeds and then trying to get the phase-lock loop system to translate them. The maximum code speed that the phase-lock loop unit could translate was 40 words per minute (WPM) on an IBM-compatible 8088 computer. Because it is extremely difficult to get the phase-lock loop locked onto a rapid signal, is not known if the 40 WPM maximum speed is a limitation of the phase-lock loop chip itself. By running diagnostic programs on the 40 WPM signal, it was determined that the program was getting about 400 samples in the space of a dit. In contrast, the DSP system, using an identical means of translation was able to accurately translate Morse code signals when it was receiving less than 3 samples per dit. Therefore, one would expect that the translation program is not the limiting factor in the speed of the phase-lock loop design.

Speed Test: DSP Board. The maximum translation speed obtainable using the DSP board as the method of detection was determined to be 60 WPM on an IBM-compatible 80386 computer using a 32-point FFT. At this code speed, the translation program was able to obtain 1 - 2 samples per dit and could translate the signal with 100% accuracy (it should be noted that the signal was very clear and strong). At a speed of 70 WPM, the DSP system was able to attain an accuracy of only about 75%.

It is possible that higher speeds could be reached by using an FFT with less points (i.e. a 16-point FFT). This was not tested, but I suspect that it would not result in an improvement in code speed since the limiting factor seems to be the speed of the computer running the translation program, not the speed of the DSP.

DSP Multiple Signal Test. Because the DSP system can distinguish between signals theoretically as close as 250 Hz, the DSP-based translation system ought to be able to detect and translate multiple Morse code signals, providing these signals are properly separated in frequency. To test this, a recording was made of two Morse code signals differing in frequency by 800 Hz and playing simultaneously. The code speeds were 16 WPM and 13 WPM and the intensity of the 16 WPM signal was slightly less than that of the 13 WPM signal. Under the control of the program CODE.EXE, these signals were sent to the DSP board. The DSP system was able to translate both signals with no errors.

The translation program was also modified to interpret three signals simultaneously. The tape I made to do this, however, was apparently not clear enough and consequently the translation was garbled. I suspect that the problem is due to the present system's sensitivity to moderate levels of background noise (see related discussion below).

DSP System: Translation of Signals With Noise. It was desired to see whether the current DSP system could translate Morse code signals taped from actual broadcasts containing background noise. However, on the day the tape of real Morse code signals was prepared, atmospheric conditions were poor and consequently there was such a high level of background noise that it was difficult to hear some of the signals when the tape was replayed. The DSP system was unable to translate these signals, and also was unable to accurately translate signals embedded in only moderate amounts of noise.

Variable Code Speed. Originally, I thought it would be necessary for the translation program to incorporate some method of tracking Morse code signals that changed in speed as they were being sent. Therefore, I wrote a procedure to perform this tracking and inserted it in the translation program. To test this procedure, a tape of Morse code was made using a computer program capable of generating Morse code. As the computer generated the code, the speed was increased 1 WPM after each

5 - 10 elements (dits or dahs) were sent. It was found that the translation program was able to track the signal and translate it correctly.

Even though the speed tracking test was successful, the results do not seem significant since the manner in which the code speed was varied does not mirror the way the speed of a real-world signal would change. The speed of a real signal would most likely experience variations if a human was sending it. However, a person is unlikely to change the speed of the signal by more than several WPM. Such a change in signal speed is taken into account when the thresholds for the element lengths are determined. Thus, for variations in speed due to human error in sending code, auto-tracking is unnecessary. Also, for large speed changes, the change in speed is more likely to be abrupt than it is to be a linear increase or decrease as in the test. Thus it is more important to redetermine the threshold parameters completely at periodic intervals than it is to constantly update them in a smooth, tracking manner. For these reasons, the tracking procedure was deemed unimportant and was eliminated from the program.

Presently, if there is a sudden, large change in signal speed--such as those that occur when someone sending at 13 WPM is talking with someone sending at 20 WPM--the user needs to select the "recapture" option in the program in order to redetermine the threshold parameters. Consequently, the first 3 - 8 characters of the new signal will be missed. It has not been worthwhile to

write a procedure that automatically redetermines the operating parameters at periodic intervals because if the signal comes unlocked (this is likely since the two signals may be at slightly different frequencies) the system will need to be readjusted anyway before proper translation can occur.

LIMITATIONS

Noise Sensitivity. Although both the phase-lock loop and DSP board designs work properly when supplied with clean signals, performance deteriorates rapidly when a moderate level of background noise is added to the Morse code signal. This effect is more noticeable with the DSP design than with the phase-lock loop design (see above discussion).

Adjustability. Both the phase-lock loop and the DSP board designs require careful manual adjustment to get locked onto the proper frequency. This is easily done with the DSP board program by running the diagnostic routine Find Channel (see Appendix C), however, it would be more desirable to have this automated.

It is very difficult to get the phase-lock loop locked onto the frequency of the Morse code signal, especially when this signal is fairly rapid. This is because at high code speeds it is impossible to tell whether there is flicker in the LED--the sign of a poor lock. Also, if the lock is not correct the parameters determined by the translation program during start-up will be incorrect. Therefore, even if one subsequently obtains a

proper lock, the output will continue to appear garbled until the start-up parameters are redetermined. It has been pointed out that one reason the current design is so difficult to adjust is because it requires manual adjustment of the potentiometer connected to the phase-lock loop. If the Morse code signal was actually coming from a radio, one could adjust the frequency of the signal instead [7].

Microprocessor Sensitivity. When the DSP-based translation system was run on different microprocessors, it was found that performance was significantly related to the processor's speed. For example, when the DSP system was run on an 8088 computer it was only able to translate Morse code at 20 WPM, whereas when it was run on an 80386 machine it was able to translate at 60 WPM. This seems to indicate that the limiting factor in the design is the speed of the translation program being run by the host computer.

The phase-lock loop design has only been tested on an 8088 IBM-compatible computer.

CONCLUSION

Summary of Work. The work performed for this project satisfied the initial goal of constructing a functional Morse code translation system. Both the phase-lock loop and the DSP systems can accurately translate clean Morse code signals. In addition, the DSP system can accurately translate multiple

signals at high speeds and displays the potential to compete with commercial Morse code translation systems--providing a less expensive DSP becomes available. Noise considerations were not a part of this project, but clearly, if the DSP system is to prove useful, ways of dealing with unwanted noise must be found.

Ideas for Future Work. Several additions and modifications would improve the portability and performance of the current design:

* Construct a game port and/or a serial port interface for the phase-lock loop detection hardware.

* Write a fast routine to write characters to the monitor, rather than using the Turbo C putchar function.

* Optimize the computer programs. These programs are the first I have written using C; as a result, I have probably not done everything in the best way.

* Optimize the translation program by making it do more processing while the DSP is computing a FFT. Currently, the main translation program only does some minor checks during this time.

* Implement a discrete Fourier transform which could be performed for a given frequency component once the channel that the desired signal was located on had been determined. This would be faster than computing the complete spectrum.

* Improve the performance of the DSP board design when there is a high degree of background noise contained with the desired signal. One way of doing this would be to increase the number of points in the FFTs.

* Improve the capability of the DSP board design to automatically determine the channel that the signal is on, the appropriate magnitude threshold, and the duration of the various code elements.

* Use a less expensive DSP board to perform the FFTs, thus reducing the cost of this method of detection.

* Write a procedure for the phase-lock loop translation program which would continuously redetermine the operating parameters. This would allow one to adjust the phase lock loop and see the resulting output continuously on the screen, thus enabling one to tell when a proper lock had been achieved by when the output started to make sense.

* Create a buffer which would store the samples that are received when the program is not translating code. This would allow a person to see what had been received while he/she was adjusting the program's parameters.

CITED REFERENCES

1. Frohne, H.R. "CW regeneration using phase-locked loop technology." Walla Walla College, 1982.

2. Jonsson, K. "Real-time Morse code detection using fast Fourier transform multi-channel band-pass filtering." Walla Walla College, 1990.

3. JDR Microdevices. "JDR PR-1 and PR-2 user's manual." San Jose: JDR Microdevices, 1986.

4. VanScheik, W. Engineering Student, Walla Walla College. Personal Interview. College Place, Spring Quarter, 1991.

5. Amateur Radio Relay League. The ARRL Handbook for Radio Amateurs. 68th ed. Newington: ARRL, 1990.

6. Couch II, L. Digital and Analog Communication Systems. 3rd ed. New York: Macmillan, 1990.

7. Frohne, H.R. Professor of Electrical Engineering, Walla Walla College. Personal Interview. College Place, Spring Quarter, 1991.

8. Communications, Automation & Control. "D3EMU Software Emulator and Hardware Reference Manual for the [AT&T] DSP32-PC." Release 1.8. 1989 [?].

ADDITIONAL REFERENCES

AT&T. WE DSP32 and DSP32C C Language Compiler: Library Reference Manual. United States: AT&T Documentation Management, 1988.

AT&T. WE DSP32 and DSP32C C Language Compiler: User Manual. United States: AT&T Documentation Management, 1988.

AT&T. WE DSP32 and DSP32C Support Software Library: User Manual. United States: AT&T Documentation Management, 1988.

Borland International. Turbo C: Reference Guide. Version 2.0. Scotts Valley: Borland, 1988.

Borland International. Turbo C: User's Guide. Version 2.0. Scotts Valley: Borland, 1988.

Bracewell, R. The Fourier Transform and Its Applications. 2nd ed. New York: McGraw-Hill, 1986.

Dale, N. and C. Weems. Pascal. 2nd ed. Lexington: D.C. Heath, 1987.

Eggebrecht, L. Interfacing to the IBM Personal Computer. 2nd ed. Carmel: Howard Sams and Company, 1990.

Horowitz, P., and W. Hill. The Art of Electronics. 2nd ed. Cambridge: Cambridge UP, 1989.

Kochan, S. Programming in ANSI C. Indianapolis: Hayden Books, 1989.

National Semiconductor. LS/S/TTL Logic Databook. Santa Clara: National Semiconductor: 1989.

National Semiconductor. Linear Databook. Santa Clara: National Semiconductor: 1982.

APPENDIX A

INTERFACE CARD SCHEMATICS

APPENDIX A: INTERFACE CARD SCHEMATICS

The circuitry necessary to interface the phase-lock loop to a computer is shown on the following two pages.

Figure A-1 is the schematic that was sent with the JDR-PR2 interface card [3]. It was necessary to connect the chips shown in this diagram to the interface card in the spaces marked on the card. The purposes of this circuitry are to perform address decoding on the signals from the computer's data bus, to insure proper operation of the card during direct memory accessing (DMA), and to buffer data from the interface card so that it does not constantly drive the computer's data bus.

In addition to the circuitry recommended by the card's manufacturer, it was necessary to attach the additions shown in Figure A-2. This circuitry can be read from and written to, although it was only necessary to perform read operations for this project. The LS155 decoders perform address decoding using signals from the circuitry in Figure A-1. These decoders also determine whether a read or a write operation is occurring. Output from the decoders enable the LS541 buffer and the LS373 latch, whose functions are, respectively, to buffer the input from the tone decoder(s) (LM567) and to latch data off the data bus during a write operation.

Figure A-1: Manufacturer's Suggested Interface Circuitry [3]

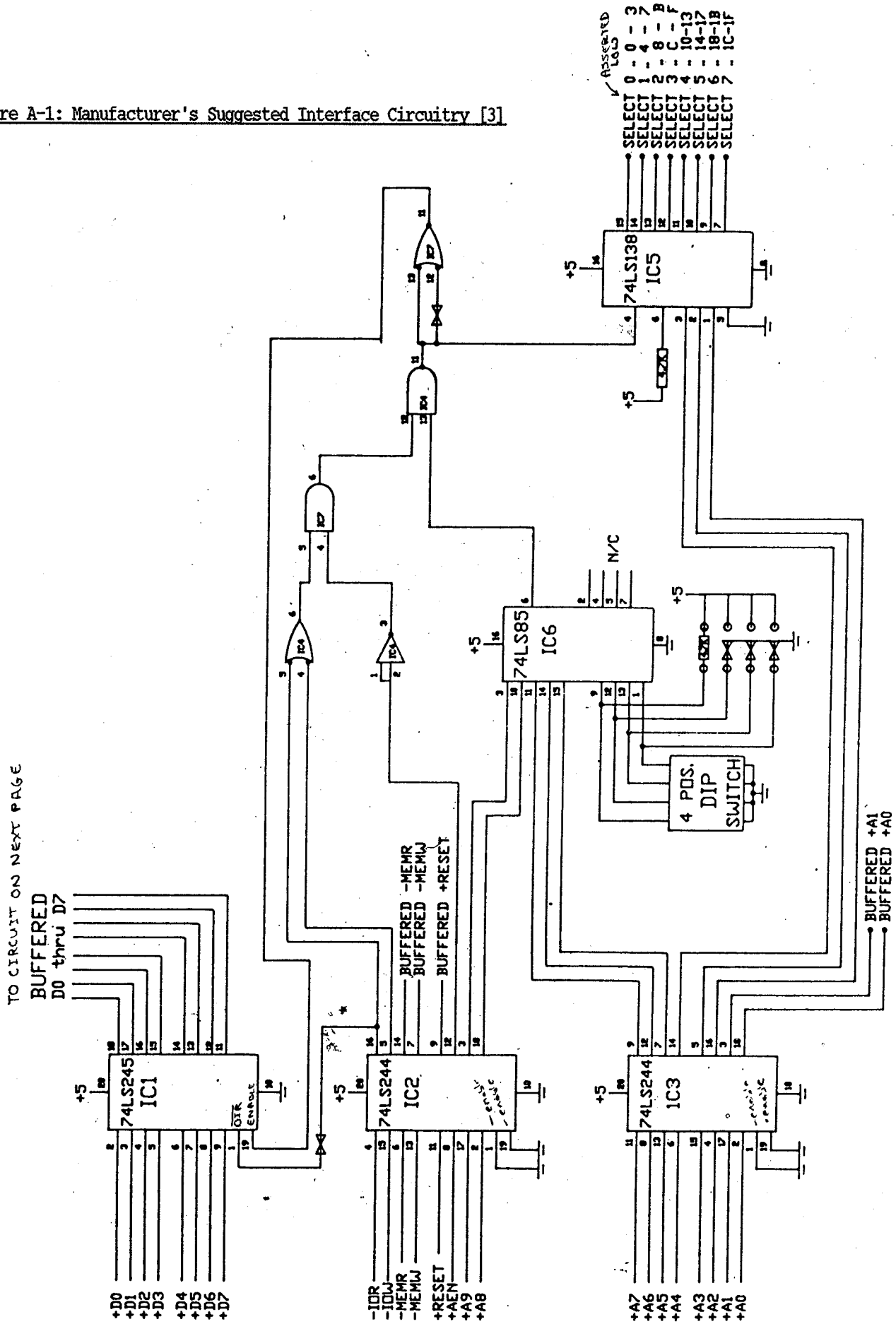
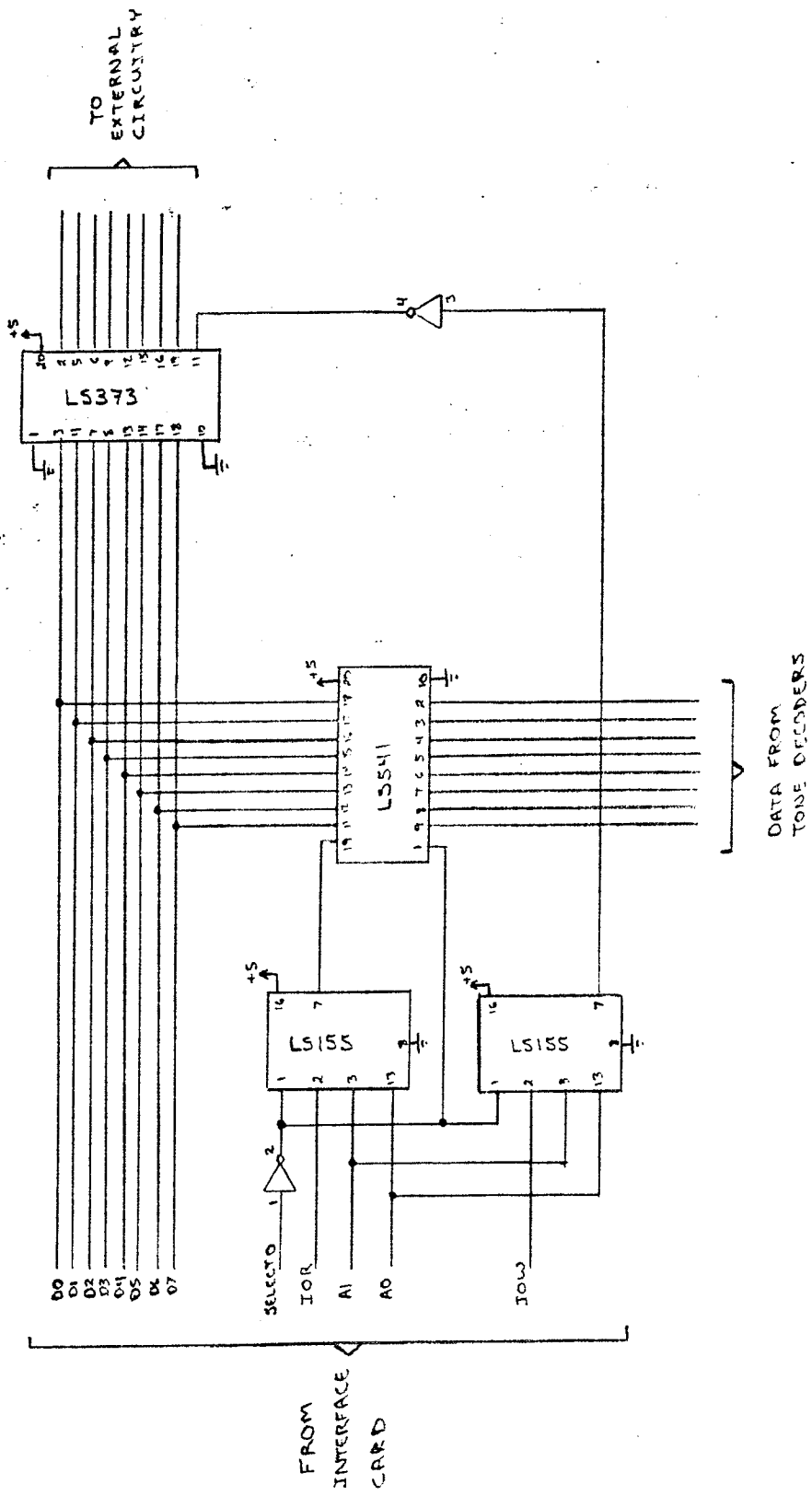


Figure A-2: Additional Interface Card Circuitry



APPENDIX B

DSP BOARD INFORMATION

APPENDIX B: DSP BOARD INFORMATION

The following tells how to use the AT&T DSP board on the computer network at Walla Walla College. More complete information on the DSP board is found in the sources listed in the bibliography of this report. Information on changes to the campus computer network configuration is obtainable from the Engineering Project Coordinator.

To install the DSP board, find an interface card address that is not occupied in the computer you will be using and set the DSP's switches to this address--as shown on page 18 of the "DSP Hardware Reference Manual" [8]. Next, plug the DSP board into an interface card slot, turn the computer on, then type "set DSP32=XXX" at the DOS prompt, where "XXX" is the hexadecimal address of the DSP--for example "230". The DSP is now ready to run programs that have already been compiled.

If it is desired to compile a new program, it is necessary to link to the DSP libraries contained on the campus computer network. This is done by typing:

"set dsp32sl=g:\engrapp\att\dsp32sl"

at the DOS prompt. Since the DSP compiler must make use of the file dsp_util.c, I found it convenient to copy this file into my own account.

When writing a program that will be used with the DSP, it is

necessary to select the "large" memory model (I wrote my programs using Turbo C Version 2.0).

A program that is to be run on the DSP must be compiled using the AT&T DSP's own compiler. To compile, type: "g:\engrapp\att\dsp32sl\bin\d3cc mycode.c -o mycode -lm -lap" Here, mycode.c is the source code to be compiled, and mycode is the name that the compiler will give to the object code and to the executable. The directives, -lm and -lap, are necessary if it is desired to include procedures in the AT&T DSP's math and applications libraries, respectively.

APPENDIX C

USER'S GUIDE TO TRANSLATION PROGRAMS

APPENDIX C: USER'S GUIDE TO TRANSLATION PROGRAMS

There are two Morse code translation programs contained in Appendix D of this report. One program, MORSE.EXE, is written for use with the phase-lock loop detection hardware and the other, CODE.EXE, is written for use with the AT&T DSP board. The operation of both of these translation programs is fairly similar and self-explanatory. Since preparing to run the program which uses the DSP board is slightly more complicated, the instructions that follow will deal with this program.

PRELIMINARIES

In order to use the translation program CODE.EXE, the AT&T DSP board must be installed in the computer on which the program will be run. This is done by plugging the DSP board in one of the expansion slots of an IBM compatible personal computer then setting the address at which the computer will look for the DSP board. For example, if the DSP board's switches are set at 230 hexadecimal, one would plug the DSP board in any one of the computer's expansion slots and type "set dsp32=230" at the DOS prompt. To determine which address the DSP board is located at, or to change the DSP board's address, see page 18 of the "Hardware Reference Manual for the DSP32-PC" [8].

MAIN MENU

After the DSP board has been properly installed, the Morse code translation program CODE.EXE can by executed by typing "code" at the DOS prompt. First, the program will display the main menu which is shown in Figure C-1. Each of the possible options that can be selected from this menu are discussed below.

Figure C-1: Main Menu

S - See Durations. Pressing "S" or "s" (none of the menu options are case sensitive) causes a subroutine to be executed that shows the magnitude and the duration of signals on a given channel that exceed a certain threshold (see G - Go for a description of how to choose different channels and set the threshold). This routine can serve as a diagnostic tool in determining how many FFTs are being performed in a dit or a dah,

and it can also be used to directly see if the proper sequence of dits and dahs is being received. Pressing "Q" or "q" during execution of this procedure returns one to the main menu.

F - Find Channel. Pressing "F" or "f" results in the execution of a subroutine that displays the channel on which the signal with the largest magnitude is being received. This routine is useful in determining the channel on which a Morse code signal is appearing in the FFT calculation. When a dit or a dah is present, the channel that is being output to the screen should correspond to the channel on which the Morse code signal is appearing in the FFT. Pressing "Q" or "q" while this subroutine is being executed will send one back to the main menu.

ESC - Quit. Pressing ESC terminates execution of the program and returns one to the DOS prompt.

G - Go. Choosing this option results in the display of a different menu. Simultaneously, the program starts locking onto the input signal by determining the length of different code elements, as well as determining the channel and the magnitude of the signal. Once the new menu has appeared, the user receives the following prompt: "Determining operating parameters, press RETURN to end." The user should wait until at least one dah and one dit have been received before pressing RETURN, otherwise the program will have no way of knowing the speed of the signal that

is being received. After RETURN has been pressed, the program will begin to translate the input signal and print the translation on the monitor. The options found in the menu that appears when Go is selected allow the user to fine-tune the parameters that were initially determined by the program. These options are discussed below.

GO MENU

When "G" or "g" is pressed at the main menu, the following menu will appear at the bottom of the screen:

R - Recapture

A - Auto Adjust Q - Quit

M - Manual Adjust

Figure C-2: Go Menu

R - Recapture. Recapture is a procedure that will completely redetermine the element lengths, channels, and peak thresholds that were determined when the Go Menu was entered. Recapture has not been implemented yet.

A - Auto Adjust. This option causes the program to repeat the determination of element lengths that was originally performed when the Go option was selected. This procedure is

useful when there is a significant change in the speed of the code being received and the program has not been able to track it.

M - Manual Adjust. Pressing "M" or "m" allows one to directly change the values of the dit, dah, and space durations. One is also able to change the number of points in the FFT and the channel of the FFT that is being used to determine the output. When Manual Adjust is selected, the user is prompted for the channel that is desired to be changed. Here, channel refers to the two signals that are displayed on the screen. Choosing channel 1 allows one to modify the parameters associated with the signal that appears in the top half of the output window, choosing channel 2 allows modification of the parameters for the signal in the lower half of the output window. The menu that appears when "M" or "m" is pressed is shown below and is self-expanatory. For instance, one can change the minimum number of

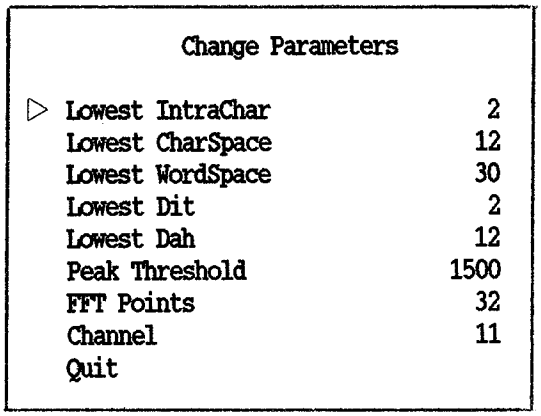


Figure C-3: Manual Adjustment Menu

samples (FFTs) in a dit from 10 to 5 by moving the cursor down to lowest dit, pressing RETURN, then typing 5 followed by RETURN.

Q - Quit. Choice of this option returns the user to the main menu.

TIPS FOR USE OF THE PROGRAM

The diagnostic tools, See Magnitudes and Durations and Find Channel, are not usually used when one desires to receive a Morse code signal. Instead, these procedures are mainly used when one is trying new types of signals or when one desires to see the type of information that the translation program is receiving from the DSP board.

Because the way the program finds the channel on which the signal is located has not been perfected, it may be desirable, when trying to translate a new signal, to first run Find Channel to determine the channel that the signal is on. Next, one should select Go, immediately going to the Manual Adjust menu and changing the channel to the value determined when running Find Channel. Lastly, one needs to run the Auto Restart option to redetermine the element durations (Auto Restart will not change the channel that you have entered at the Manual Adjust menu).

APPENDIX D

TRANSLATION PROGRAM FLOWCHARTS AND LISTINGS

APPENDIX D: TRANSLATION PROGRAM FLOWCHARTS AND LISTINGS

Listings of the following three programs appear in the pages that follow:

MORSE.C -- This program is used with the phase-lock loop interface card. It assumes that the phase-lock loop is at address 300 hexadecimal.

CODE.C -- This program is used with the AT&T DSP board. CODE.C is compiled using the Turbo C compiler, however, in order to run CODE.C the DSP based program, CODE_DSP.C, must be compiled and in the current directory.

CODE_DSP.C -- This program is actually run by the DSP board and must be compiled using the AT&T DSP's compiler.

APPENDIX D: MORSE.C

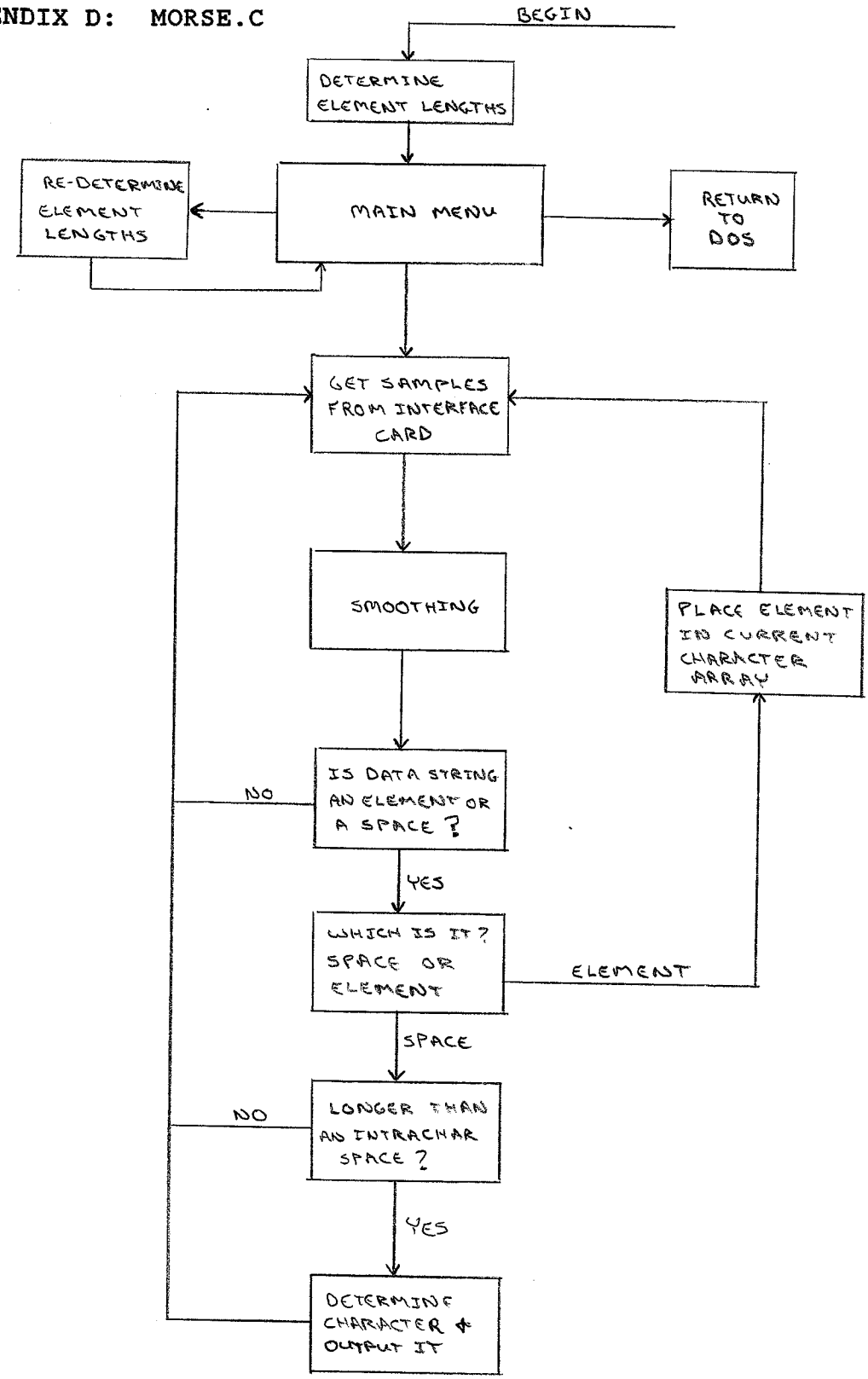


Figure D-1: Flowchart for Morse.C

```
/*************************************************************/
File Name:          Morse.C

Programmer:         Jeff McDow
Date:               11-20-90
Last Revision:      4-18-91

Written In:         Turbo C Version 2.0

Description:        This program operates in conjunction with a
                        specially constructed PC interface card
                        containing a phase-lock loop (L567 tone decoder)
                        to translate a Morse code signal in real time.

Input:              Input consists of decimal numbers read from the
                        interface card (address 300 hexadecimal).  A
                        nonzero sample indicates that no Morse code
                        signal is present. A sample value of zero
                        indicates that a Morse code element is present.

Output:             The translation of the Morse code signal (pieced
                        together from the samples) is output to the PC
                        monitor.

Limitations:        This program performs its desired function,
                        however, it is the first program that the
                        programmer had written in C and consequently
                        may not be optimum.
/*************************************************************/

#include <dos.h>
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>

#define DITSCALE 0.3          /* lowest_dit = dit*ditscale             */
#define DAHSCALE 0.55         /* lowest_dah = dah - dah*dahscale       */
#define ILSPACE 0.3           /* lowest_intrachar = dit*ilspace        */
#define CUSPACE 0.4           /* highest_charspace = dah + dah*cuspace */
#define IHSPACE 0.2           /* highest_intrachar = dit + dit*ihspace */


/******************  Function Declarations  ***************/

void out_char(int letter[]);
void recapture();
void new_line_check();
void setup();

/******************  Global Variables  ********************/

unsigned int f, off;
int kbhit(), letter[7], counter, i, on, wherex(), wherey();
int lowest_dah, lowest_dit;
int lowest_intrachar, highest_intrachar;
int highest_charspace, inbetween, last_ditdah, key;

/**************** Function Definitions *********************/
/***********************************************************/
/***********************************************************/
Name:               out_char
```

```
Called By:        main

Input:            Input to the function is a 7-element array
                         containing 1's, 2's which represent the dits and
                         dahs, respectively, of a Morse code character.
                         If the character is not 7 elements long, the
                         remainder of the array contains 0's.

Output:           Output consists of writing the character
                         represented by the input array, letter, to
                         the computer's monitor.

Description:      This function determines the identity of
                         an array of dits and dahs representing a
                         Morse code character by treating the array
                         as a base 3 number then converting it to a base ten
                         number and entering that value into a search tree
                         containing all possible letters.  The result is
                         printed on the computer's monitor.
/*************************************************************/
void out_char(int letter[])

{
int i, letout, done;
int aa,bb,cc,dd,ee,ff,gg,hh,ii,jj,kk,ll,mm,nn,oo,pp,
    qq,rr,ss,tt,uu,vv,ww,xx,yy,zz,quest,period,comma,SK,AR,DN,BT,
    n1,n2,n3,n4,n5,n6,n7,n8,n9,n0;

aa=7; bb=41; cc=50; dd=14; ee=1; ff=49; gg=17; hh=40; ii=4; jj=79;
kk=23; ll=43; mm=8; nn=5; oo=26; pp=52; qq=71; rr=16; ss=13; tt=2;
uu=22; vv=67; ww=25; xx=68; yy=77; zz=44; n0=242; n1=241; n2=238;
n3=229; n4=202; n5=121; n6=122; n7=125; n8=134; n9=161; comma=692;
quest=400; period=637; AR=151; SK=634; DN=149; BT=203;


/*  Convert letter to an integer value */

letout = 0;
done = 1;

while (done == 1)
  {
  if (letter[0]==0) break;
  else if (letter[0]==1) letout = 1;
  else letout = 2;

  if (letter[1]==0) break;
  else if (letter[1]==1) letout = letout + 3;
  else letout = letout + 6;

  if (letter[2]==0) break;
  else if (letter[2]==1) letout = letout + 9;
  else letout = letout + 18;

  if (letter[3]==0) break;
  else if (letter[3]==1) letout = letout + 27;
  else letout = letout + 54;

  if (letter[4]==0) break;
  else if (letter[4]==1) letout = letout + 81;
  else letout = letout + 162;
```

```
  if (letter[5]==0) break;
  else if (letter[5]==1) letout = letout + 243;
  else letout = letout + 486;

  if (letter[6]==0) break;
  else if (letter[6]==1) letout = letout + 729;
  else letout = letout + 1458;

  done = 0;
  }


/*  Find What the Character is and Output it */

if (letout<=pp)
  if (letout<=gg)
    if (letout<=nn)
      if (letout<=ee)
            {
            if (letout==ee) putchar('e');
            else putchar(' ');
            }
      else
            {
            if (letout==tt) putchar('t');
            else if (letout==ii) putchar('i');
            else if (letout==nn) putchar('n');
            }
    else
      if (letout<=ss)
            {
            if (letout==aa) putchar('a');
            else if (letout==ss) putchar('s');
            else if (letout==mm) putchar('m');
            }
      else
            {
            if (letout==dd) putchar('d');
            else if (letout==rr) putchar('r');
            else if (letout==gg) putchar('g');
            }
  else
    if (letout<=hh)
      if (letout<=kk)
            {
            if (letout==uu) putchar('u');
            else putchar('k');
            }
      else
            {
            if (letout==oo) putchar('o');
            else if (letout==hh) putchar('h');
            else if (letout==ww) putchar('w');
            }
    else
      if (letout<=zz)
            {
            if (letout==ll) putchar('l');
            else if (letout==bb) putchar('b');
            else if (letout==zz) putchar('z');
            }
```

```
      else
            {
            if (letout==cc) putchar('c');
            else if (letout==ff) putchar('f');
            else if (letout==pp) putchar('p');
            }
else
  if (letout<=AR)
    if (letout<=jj)
      if (letout<=xx)
            {
            if (letout==xx) putchar('x');
            else putchar('v');
            }
      else
            {
            if (letout==qq) putchar('q');
            else if (letout==yy) putchar('y');
            else if (letout==jj) putchar('j');
            }
    else
      if (letout<=n7)
            {
            if (letout==n7) putchar('7');
            else if (letout==n5) putchar('5');
            else if (letout==n6) putchar('6');
            }
      else
            {
            if (letout==n8) putchar('8');
            else if (letout==DN) putchar('/');
            else if (letout==AR) putchar('+');
            }
  else
    if (letout<=n2)
      if (letout<=n4)
            {
            if (letout==n9) putchar('9');
            else putchar('4');
            }
      else
            {
            if (letout==n3) putchar('3');
            else if (letout==n2) putchar('2');
            else if (letout==BT) putchar('-');
            }
    else
      if (letout<=quest)
            {
            if (letout==n1) putchar('1');
            else if (letout==n0) putchar('0');
            else if (letout==quest) putchar('?');
            }
      else
            {
            if (letout==period) putchar('.');
            else if (letout==comma) putchar(',');
            else if (letout==SK) putchar('#');
            }
}
/*************************************************************/
```

```
/*************************************************************/
Name:              recapture

Called By:         main

Input:             None

Output:            None

Description:       This function determines the length of a dah
                          by taking successive samples from the interface
                          card, adding them, and determining the longest
                          time the Morse code signal is on.  This length
                          is called a dah, and from it all the other Morse
                          code parameters are calculated.  This function is
                          exited by pressing any key.
/*************************************************************/
void recapture ()

{
int longest, dit, dah;


/* Determine the length of the longest Morse code element,
   stop when the user presses any key.                    */

while (kbhit()==0)
  {
  f = inportb(0x300);
  while (f==0)
    {
    on = on++;
    f = inportb(0x300);
    }
  if (on > longest)
  longest = on;
  on = 0;
  }

/*  Determine other Morse code parameters    */

dah = longest;
dit = longest/3;
lowest_dah = dah - dah*DAHSCALE;
lowest_dit = dit*DITSCALE;
lowest_intrachar = dit*ILSPACE;
highest_intrachar = dit + dit*IHSPACE ;
highest_charspace = dah + dah*CUSPACE;

longest = wherex();
dah = wherey();
gotoxy(60,1);
printf("dit %d ",dit);
gotoxy(longest,dah);

}
/*************************************************************/
/*************************************************************/
Name:              new_line_check
```

```
Called By:        main

Input:            None

Output:           None

Description:      This function determines when the output
                        to the monitor needs to be put on a new line.
                        The function also clears space ahead of the
                        output so that the current output is not
                        writing over the top of previous output.
/*************************************************************/
void new_line_check()

{
int posx, posy;

posx = wherex();          /** Check to see if a new line is needed **/
if (posx > 60)
  {
  posy = wherey();        /** Clear a space ahead **/
  if (posy < 19)
    {
    gotoxy(1,posy+3);
    clreol();
    gotoxy(posx,posy);
    }
  else
    {
    gotoxy(1,posy-19+4);
    clreol();
    gotoxy(posx,posy);
    }
  if (posy == 21)         /** Start a new line **/
    {
    gotoxy(1,3);
    printf("\n");
    }
  else
    printf("\n");
  }
}
/*************************************************************/
/*************************************************************/
Name:             setup

Called By:        main

Input:            None

Output:           None

Description:      This function sets up the screen by drawing
                        the menus.
/*************************************************************/
void setup()

{
int iset, wpm;
```

```
clrscr();

wpm = 13;

gotoxy(60,2);
printf("wpm = %d",wpm);
gotoxy(5,24);
printf("r - Recapture");
gotoxy(60,24);
printf("q - Quit");

gotoxy(1,22);                          /** Draw Horizontal Lines **/
for (iset=1; iset<81; iset++)
  putchar(196);

gotoxy(1,3);                           /** Draw Horizontal Lines **/
for (iset=1; iset<81; iset++)
  putchar(196);

gotoxy(1,4);

}
/*************************************************************
                    MAIN PROGRAM
*************************************************************/
main()

{
setup();                               /** Initialize Screen **/
recapture();                           /** Determine Element Lengths **/

on = 0;
off = 0;
f = inportb(0x300);
counter = 0;

while (key!=113)                       /** Repeat until 'q' is pressed **/
  {
  if (kbhit()!=0)
    key = getch();
  if ((key==114) || (key==82))         /** If user presses 'r' or 'R' **/
    recapture();                       /** then redetermine parameters **/

  if (f==0)                            /** If Morse code signal is present ...**/
    {                                  /** then process it                  **/
    on = 0;
    while (f==0)                       /** While signal is on ... **/
      {
      on = on++;                       /** increment length of signal **/
      f = inportb(0x300);              /** get a new sample from interface **/
      }
    if (on > lowest_dit)               /** If length was longer than lowest dit **/
      {
      if (inbetween < lowest_intrachar)   /** if last space was not at least an intrachar **/
            last_ditdah = last_ditdah + inbetween + on; /** then add length to last element **/
      else
            if (inbetween < highest_intrachar)   /** if last space is an intracharacter space... **/
              {
              if (counter > 7)             /** if character array is full, print error msg **/
                {
                printf("  ERROR  ");
```

```
                out_char(letter);
                for (i = 1; i <= 7; i++)
                  letter[i-1] = 0;
                counter = 0;
                }
              else                          /** otherwise find if element is dit or dah **/
                {
                if (last_ditdah > lowest_dah)
                  letter[counter] = 2;
                else
                  letter[counter] = 1;
                last_ditdah = on;           /** store element in character array ... **/
                counter = counter++;        /** and get ready for next element       **/
                }
              }
            else
              {
              if (last_ditdah > lowest_dah)  /** if last space was word or interchar... **/
                letter[counter] = 2;         /** store element in character array **/
              else
                letter[counter] = 1;
              last_ditdah = on;
              counter = 0;
              out_char(letter);              /** print character out **/
              for (i = 1; i <= 7; i++)       /** reinitialize character array **/
                letter[i-1] = 0;
              if (inbetween > highest_charspace)  /** if last space was a word space **/
                {
                putchar(' ');                     /** then output a blank space and ... **/
                new_line_check();                 /** check to see if a new line is needed **/
                }
              }
            inbetween = 0;                  /** reinitialize spacing counter **/
            }
        else
            inbetween = inbetween + on;
      }
                                                  /** if a space is being received **/
    if (f!=0)
      {
      off = 0;
      while (f!=0)            /** increment space counter and get a new sample **/
        {
        off = off++;
        f = inportb(0x300);
        }
      inbetween = inbetween + off;
      }
    }
  }
```

APPENDIX D: CODE.C

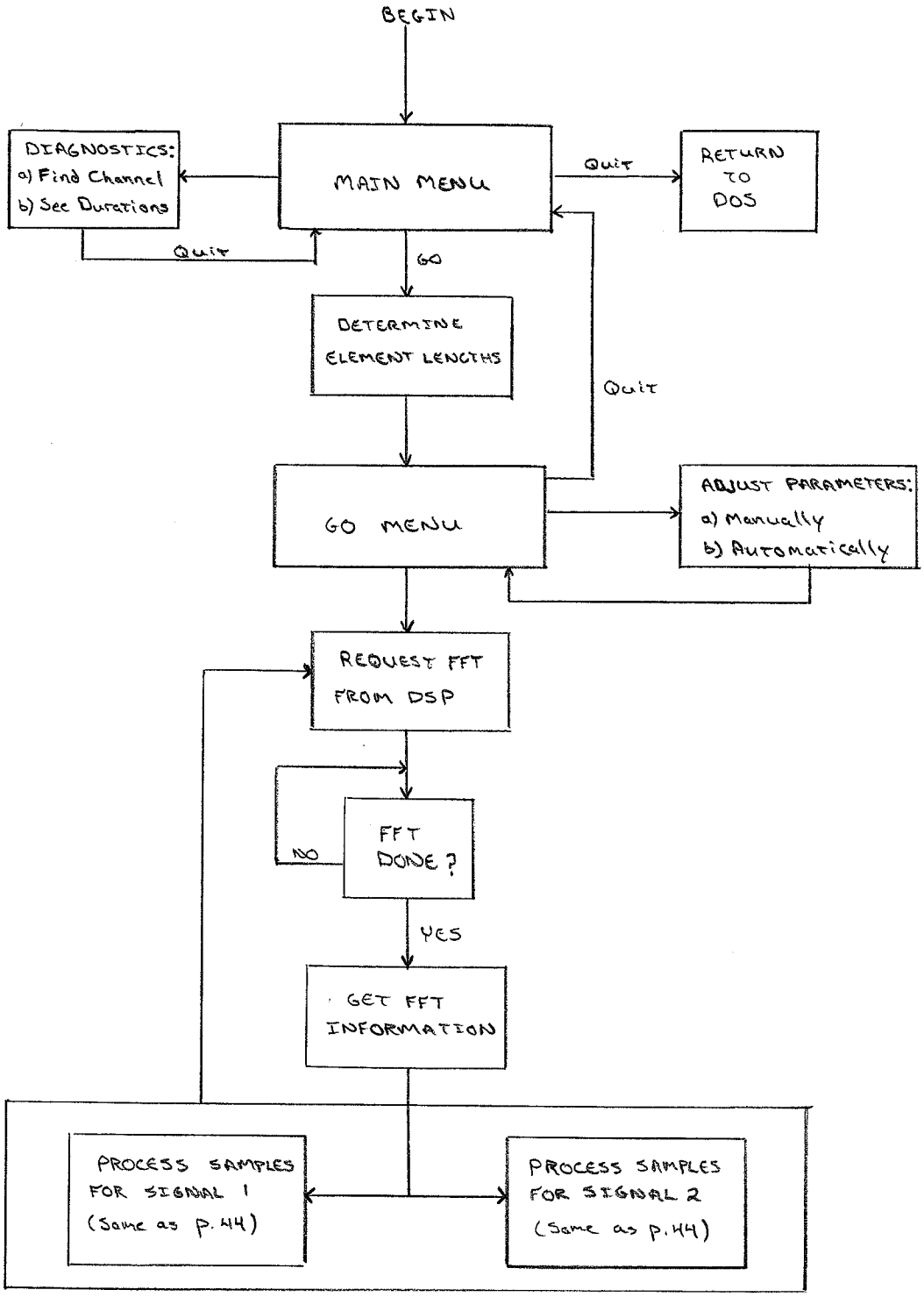


Figure D-2: Flowchart for Code.C

```
/************************************************************
Filename:       CODE.C
Programmer:     Jeff McDow
Date:           4-29-91
Last Revision:  5-14-91

Written In:     Turbo C Version 2.0

Description:    This program has the capability of translating
                two Morse code signals simultaneously and in
                real-time.

Input:          Input is received from the AT&T Digital Signal
                Processing board which samples the Morse code
                and does a FFT on it under the control of the
                program CODE_DSP.C.  Input consists of the magnitude
                of a specified bin number in the FFT.

Output:         The translation of the Morse code signal(s)
                (pieced together from the samples) is output
                to the computer's monitor.

Limitations:    This program performs its specified function.  The
                options "See Magnitudes and Durations", "Find
                Channel", "Auto Adjust", and "Recapture" are not
                implemented in this version of CODE.C.  This program
                must be run with the compiled version of the DSP
                based program CODE_DSP.C in the same directory.
************************************************************/

#include <dos.h>
#include <alloc.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <conio.h>

#define DITSCALE 0.1          /* lowest_dit = dit*ditscale       */
#define DAHSCALE 0.35         /* lowest_dah = dah - dah*dahscale */
#define ILSPACE 0.1           /* lowest_intrachar = dit*ilspace  */
#define LWSPACE 1.5           /* lowest_word = dah + dah*cuspace */
#define LCSPACE 0.2           /* lowest_char = dit + dit*ihspace */

#define SUX 1                 /* SUX, SUY, SLX, SLY define the   */
#define SUY 4                 /* corners of the output window.   */
#define SLX 80
#define SLY 21
#define MUX 1                 /* MUX, MUY, MLX, MLY define the   */
#define MUY 23                /* corners of the main menu.       */
#define MLX 80
#define MLY 25
#define WUX 31                /* WUX, WUY, WLX, WLY define the   */
#define WUY 5                 /* corners of the Manual Adjustment*/
#define WLX 61                /* menu.                           */
#define WLY 19

/********************** Function Declarations ************/

void exit_routine();
void download ();
void init ();
```

```
void do_fft ();
void out_char(int letter[], int *xloc, int yloc);
void new_line_check(int *xloc, int *yloc);
void setup();
void process_off(int *inbetween, int total);
void process_on(int letter[], int *inbetween, int *lddah,
                                        int lowest_intrachar, int lowest_char, int lowest_word
,
                                        int *counter, int lowest_dit, int total, int *xloc,
                                        int *yloc, int lowest_dah);
void get_character();
void manual_handler(int *lowest_intrachar,int *lowest_char,int *lowest_word,
                                                int *lowest_dit,int *lowest_dah,int *peak_
thresh,
                                                int *channel);
void see_handler();
void go_handler();
void find_handler();

/***************** Global Variables **********************/

int letter1[7], letter2[7], total1=0, total2=0, last1=0, last2=0;
int inbetween1=0, inbetween2=0, value1=0, value2=0, do_1=0, do_2=0;
int f, off, do_go, do_see, do_manual, do_find, do_recap, do_get;
int kbhit(), on, wherex(), wherey(), letcnt1=0, letcnt2=0;
int lowest_dah1, lowest_dah2, lowest_dit1, lowest_dit2;
int lowest_intrachar1, lowest_intrachar2, lowest_char1, lowest_char2;
int lowest_word1, lowest_word2, lddah1=0, lddah2=0;
int peaknum, wpm, dit, fftpts, channel1, channel2;
int peak_thresh1, peak_thresh2, xloc1, yloc1, xloc2, yloc2;
char param_menu[31*15*2], clear_display[18*80*2], clear_menu[80*3*2];
char main_menu[80*3*2], go_menu[80*3*2], save_pmenu[31*15*2];
long flag, n, mag1, mag2, chan1, chan2;

/*************************************************************/
/***************** Function Definitions **********************/
/*************************************************************
Name:         exit_routine

Called By:    main

Description:  This function clears the computer's monitor
                  when execution of this program is completed.

Global Variables Changed:    None
*************************************************************/
 void exit_routine()
{
  clrscr();
  exit(1);
}
/*************************************************************/
/*************************************************************
Name:         download

Called By:    init, manual_handler

Description:  Values of the variables fftpts, channel1,
                  and channel2 are loaded into variables in
                  the DSP board.
```

```
Global Variables Changed:    None
*************************************************************/

void download ()

{
dsp_dl_int (n, fftpts);
dsp_dl_int(chan1, channel1);
dsp_dl_int(chan2, channel2);
}

/*************************************************************/
/*************************************************************
Name:         init

Called By:    main

Calls:        download

Description:  Initializes the DSP and assigns addresses
                  to DSP variables using "find_label_name".

Globals Changed:   None

Adapted From Kristjan Jonsson's program CW_DET1.C
*************************************************************/

void init()

{
  if (!dsp_dl_exec ("code _dsp"))
            {
            printf("cannot load code _dsp to dsp32\n\n");
            exit(1);
            }

  flag = find_label_name ("flag");
  chan1 = find_label_name ("chan1");
  chan2 = find_label_name ("chan2");
  mag1 = find_label_name ("mag1");
  mag2 = find_label_name ("mag2");
  n = find_label_name ("n");

  dsp_run();

  download();
}
/*************************************************************/
/*************************************************************
Name:         do_fft

Called By:    get_character

Description:  This function tells the DSP to perform a
                  FFT, waits for the FFT to be performed, then
                  loads the magnitudes of the desired bin numbers
                  of the FFT into the variables value1 and
                  value2.  These magnitudes are then converted
                  into 1's or 0's depending on whether or not
                  they exceed a certain pre-specified threshold.
```

```
                Communication with the DSP is mediated by the
                variable flag.  A flag of 1 tells the DSP to
                perform a FFT.  A flag of 0 tells the PC that
                the DSP has finished doing the FFT.

Globals Changed:  value1, value2
*************************************************************/

void do_fft()

{
  int peak_value, channel;

   dsp_dl_int (flag,1);
   while(dsp_up_int(flag));
   value1 = dsp_up_int(mag1);
   value2 = dsp_up_int(mag2);
   dsp_dl_int (flag,1);
   if (value1 > peak_thresh1)
            value1 = 0;
   else
            value1 = 1;
   if (value2 > peak_thresh2)
            value2 = 0;
   else
            value2 = 1;

}
/*************************************************************/
/*************************************************************
Name:         out_char

Called By:    process_on

Description:  This function determines the identity of an
                array of 1's and 2's representing a Morse
                code character (the array is padded with
                zeros).  The determination is done by converting
                the 1's and 2's in the array to a base 10 number,
                finding the corresponding alphanumeric character by
                entering the base 10 number in a search tree, and
                printing the character to the monitor.

Globals Changed:  The x-location that is passed to out_char
                        (either xloc1 or xloc2) is modified.

*************************************************************/

void out_char(int letter[], int *xloc, int yloc)

{
int letout=0, done=1;
int aa,bb,cc,dd,ee,ff,gg,hh,ii,jj,kk,ll,mm,nn,oo,pp,
    qq,rr,ss,tt,uu,vv,ww,xx,yy,zz,quest,period,comma,SK,AR,DN,BT,
           n1,n2,n3,n4,n5,n6,n7,n8,n9,n0;

aa=7; bb=41; cc=50; dd=14; ee=1; ff=49; gg=17; hh=40; ii=4; jj=79;
kk=23; ll=43; mm=8; nn=5; oo=26; pp=52; qq=71; rr=16; ss=13; tt=2;
uu=22; vv=67; ww=25; xx=68; yy=77; zz=44; n0=242; n1=241; n2=238;
n3=229; n4=202; n5=121; n6=122; n7=125; n8=134; n9=161; comma=692;
quest=400; period=637; AR=151; SK=634; DN=149; BT=203;
```

```
/*  Convert letter to an integer value */

while (done == 1)
  {
  if (letter[0]==0) break;
  else if (letter[0]==1) letout = 1;
  else letout = 2;

  if (letter[1]==0) break;
  else if (letter[1]==1) letout = letout + 3;
  else letout = letout + 6;

  if (letter[2]==0) break;
  else if (letter[2]==1) letout = letout + 9;
  else letout = letout + 18;

  if (letter[3]==0) break;
  else if (letter[3]==1) letout = letout + 27;
  else letout = letout + 54;

  if (letter[4]==0) break;
  else if (letter[4]==1) letout = letout + 81;
  else letout = letout + 162;

  if (letter[5]==0) break;
  else if (letter[5]==1) letout = letout + 243;
  else letout = letout + 486;

  if (letter[6]==0) break;
  else if (letter[6]==1) letout = letout + 729;
  else letout = letout + 1458;

  done = 0;
  }

if (do_1)
  window(1,4,80,12);
else
  window(1,13,80,21);

gotoxy(*xloc,yloc);

/*  Find What the Character is and Output it */

if (letout<=pp)
  if (letout<=gg)
    if (letout<=nn)
      if (letout<=ee) {
            if (letout==ee) putchar('e');
            else putchar(' '); }
      else {
            if (letout==tt) putchar('t');
            else if (letout==ii) putchar('i');
            else if (letout==nn) putchar('n'); }
    else
      if (letout<=ss) {
            if (letout==aa) putchar('a');
            else if (letout==ss) putchar('s');
            else if (letout==mm) putchar('m');}
      else {
            if (letout==dd) putchar('d');
```

```
            else if (letout==rr) putchar('r');
            else if (letout==gg) putchar('g');}
    else
      if (letout<=hh)
        if (letout<=kk) {
            if (letout==uu) putchar('u');
            else putchar('k');}
        else {
            if (letout==oo) putchar('o');
            else if (letout==hh) putchar('h');
            else if (letout==ww) putchar('w');}
      else
        if (letout<=zz) {
            if (letout==ll) putchar('l');
            else if (letout==bb) putchar('b');
            else if (letout==zz) putchar('z');}
        else {
            if (letout==cc) putchar('c');
            else if (letout==ff) putchar('f');
            else if (letout==pp) putchar('p');}
  else
    if (letout<=AR)
      if (letout<=jj)
          if (letout<=xx)
                {
                if (letout==xx) putchar('x');
                else putchar('v');
                }
          else
                {
                if (letout==qq) putchar('q');
                else if (letout==yy) putchar('y');
                else if (letout==jj) putchar('j');
                }
        else
          if (letout<=n7)
                {
                if (letout==n7) putchar('7');
                else if (letout==n5) putchar('5');
                else if (letout==n6) putchar('6');
                }
          else
                {
                if (letout==n8) putchar('8');
                else if (letout==DN) putchar('/');
                else if (letout==AR) putchar('+');
                }
    else
      if (letout<=n2)
          if (letout<=n4)
                {
                if (letout==n9) putchar('9');
                else putchar('4');
                }
          else
                {
                if (letout==n3) putchar('3');
                else if (letout==n2) putchar('2');
                else if (letout==BT) putchar('-');
                }
      else
```

```
            if (letout<=quest)
                    {
                    if (letout==n1) putchar('1');
                    else if (letout==n0) putchar('0');
                    else if (letout==quest) putchar('?');
                    }
            else
                    {
                    if (letout==period) putchar('.');
                    else if (letout==comma) putchar(',');
                    else if (letout==SK) putchar('#');
                    }

/** update the position of the text on the screen **/

*xloc = *xloc + 1;
window(1,1,80,25);

}  /* End Out_Char */

/*************************************************************/
/*************************************************************
Name:         new_line_check

Called By:    process_on

Description:  This function determines if it is necessary to
              start a new line in the output window.  The
              function also clears a line that is ahead of
              the current line.

Globals Changed:  Both the x and y screen coordinates that are
                  passed to this function may be changed.
                  (xloc1, xloc2, yloc1, yloc2)
*************************************************************/

void new_line_check(int *xloc, int *yloc)

{
if (do_1)
  window(1,4,80,12);
else
  window(1,13,80,21);

*xloc = *xloc + 1;
gotoxy(*xloc,*yloc);

if (*xloc > 30)
  {
  if (*yloc < 7)
          {
          gotoxy(1,*yloc+3);
          clreol();
          gotoxy(*xloc,*yloc);
          }
  else
          {
          gotoxy(1,*yloc-7+1);
          clreol();
          gotoxy(*xloc,*yloc);
```

```
            }
    }
  if (*xloc > 60)
    {
    if (*yloc == 9)
            gotoxy(1,1);
    printf("\n");
    }

  *xloc = wherex();
  *yloc = wherey();

  window(1,1,80,25);

  }
  /*************************************************************/
  /*************************************************************
  Name:         setup

  Called By:    main

  Description:  This function sets up the user screen and draws
                  and stores all of the possible menus.

  Globals Changed: None
  *************************************************************/
  void setup()

  {
  int i;

  gettext(1,4,80,21,clear_display);
  gettext(1,23,80,25,clear_menu);

  /* Draw Manual Adjustment Window */

  gotoxy(1,1);
  putchar(201);
  for (i=2; i<15; i++)
    {
    gotoxy(1,i);
    putchar(186);
    }
  gotoxy(1,15);
  putchar(200);
  gotoxy(2,1);
  for (i=1; i<30; i++)
    putchar(205);
  putchar(187);
  for (i=2; i<15; i++)
    {
    gotoxy(31,i);
    putchar(186);
    }
  gotoxy(31,15);
  putchar(188);
  gotoxy(2,15);
  for (i=1; i<30; i++)
    putchar(205);
  gotoxy(7,3);
  printf("Change Parameters ");
```

```
gotoxy(3,5);
putchar(16);
gotoxy(5,5);
printf("Lowest Intrachar");
gotoxy(5,6);
printf("Lowest Char_Space");
gotoxy(5,7);
printf("Lowest WordSpace");
gotoxy(5,8);
printf("Lowest Dit");
gotoxy(5,9);
printf("Lowest Dah");
gotoxy(5,10);
printf("Peak Threshold");
gotoxy(5,11);
printf("FFT Points");
gotoxy(5,12);
printf("Channel");
gotoxy(5,13);
printf("Quit");

gettext(1,1,31,15,param_menu);


/** Draw Main Menu **/

gotoxy(5,23);
printf("F - Find Channel");
gotoxy(5,25);
printf("S - See Durations");
gotoxy(60,23);
printf("G - Go");
gotoxy(60,25);
printf("ESC - Exit");

gettext(1,23,80,25,main_menu);
puttext(1,23,80,25,clear_menu);

/** Draw Go Menu **/

gotoxy(5,23);
printf("R - Recapture");
gotoxy(60,24);
printf("Q - Quit");
gotoxy(5,24);
printf("A - Auto Adjust");
gotoxy(5,25);
printf("M - Manual Adjust");

gettext(1,23,80,25,go_menu);

clrscr();

/** Draw Main Screen **/

gotoxy(1,22);
for (i=1; i<81; i++)
  putchar(196);

gotoxy(1,3);
for (i=1; i<81; i++)
```

```
  putchar(196);

gotoxy(60,1);
printf("Revision 9");

puttext(1,23,80,25,main_menu);
gotoxy(1,4);

}
/*************************************************************/
/*************************************************************
Name:         process_off

Called By:    get_character

DESCRIPTION:  After a string of samples has been received
from the DSP board indicating that the Morse code signal was
off, the total number of samples in this string is added to
the running total for the current "space".  Note that the current
"space" may consist of more than just the current string of "off"
values do to smoothing.

Globals Changed:  The total of consecutive "offs" that are passed
                          (inbetween1 or inbetween2) is modified.
*************************************************************/
void process_off(int *inbetween, int total)
{

*inbetween = *inbetween + total;

}
/*************************************************************/
/*************************************************************
Name:         process_on;

Called By:    get_character

Calls:        out_char, new_line_check

DESCRIPTION:  After a string of "on" samples are
received from the DSP board, process_on determines whether
this string is a dit, a dah, or a glitch.  Furthermore, if the
string completes a dit or a dah, process_on checks to see if a
character needs to be printed out and if a new line needs to be
started.

Globals Changed:  None other than those passed by address in the
                          function call.
*************************************************************/

void process_on(int letter[], int *inbetween, int *lddah,
                                              int lowest_intrachar, int lowest_char, int lowest_word
,
                                              int *counter, int lowest_dit, int total, int *xloc,
                                              int *yloc, int lowest_dah)


{
int i;

if (total > lowest_dit)
  {
```

```
  if (*inbetween < lowest_intrachar)
          *lddah = *lddah + *inbetween + total;
  else
          if (*inbetween < lowest_char)
            {
            if (*counter > 7)
                    {
                    printf("  ERROR  ");
                    out_char(letter,&(*xloc),*yloc);
                    for (i = 1; i <= 7; i++)
                      letter[i-1] = 0;
                    *counter = 0;
                    }
            else
                    {
                    if (*lddah > lowest_dah)
                      letter[*counter] = 2;
                    else
                      letter[*counter] = 1;
                    *lddah = total;
                    *counter = *counter + 1;
                    }
            }
          else
            {
            if (*lddah > lowest_dah)
                    letter[*counter] = 2;
            else
                    letter[*counter] = 1;
            *lddah = total;
            *counter = 0;
            out_char(letter,&(*xloc),*yloc);
            for (i = 1; i <= 7; i++)
                    letter[i-1] = 0;
            if (*inbetween > lowest_word)
                    {
                    putchar(' ');
                    new_line_check(&(*xloc),&(*yloc));
                    }
            }
  *inbetween = 0;
  }
else
  *inbetween = *inbetween + total;
}

/*************************************************************/
/*************************************************************
Name:         get_character

Called By:    go_handler

Calls:        do_fft, process_on, process_off

DESCRIPTION:  This function checks to see if the user wants to
manually adjust the operating parameters or stop the program
from translating Morse code.  If not, get_character receives
samples and if they are all of the same value, increments a counter.
When a change in the sample value (1 to 0 or 0 to 1) is received, the
current counter is sent to process_on or process_off depending
on what the values of the samples in the previously received
```

```
string were.

Globals Changed: do_manual, do_go, do_get, value1, value2,
                        last1, last2, total1, total2
*************************************************************/

void get_character()

{
int key=0;

do_get = 1;
do_fft();

while (do_get)
  {
  do_fft();
  if (kbhit())
            {
            key = getch();
            if (((key==81)||(key==113))||((key==77)||(key==109)))
              {
              do_get = 0;
              if ((key==81)||(key==113))
                        do_go = 0;
              else
                        do_manual = 1;
              }
            }
  if (value1 != last1)
            {
            if (last1 == 0)
              {
              do_1 = 1;
              process_on(letter1,&inbetween1,&lddah1,lowest_intrachar1,
                                                  lowest_char1,lowest_word1,&letcnt1,lowest_dit1,
                                                  total1,&xloc1,&yloc1,lowest_dah1);

              total1 = 1;
              last1 = value1;
              do_1 = 0;
              }
            else
              {
              process_off(&inbetween1,total1);
              total1 = 1;
              last1 = value1;
              }
            }
  else
            total1 = total1 + 1;

  if (value2 != last2)
            {
            if (last2 == 0)
              {
              do_2 = 1;
              process_on(letter2,&inbetween2,&lddah2,lowest_intrachar2,
                                                  lowest_char2,lowest_word2,&letcnt2,lowest_dit2,
                                                  total2,&xloc2,&yloc2,lowest_dah2);

              total2 = 1;
              last2 = value2;
```

```
            do_2 = 0;
            }
          else
            {
            process_off(&inbetween2,total2);
            total2 = 1;
            last2 = value2;
            }
          }
  else
          total2 = total2 + 1;
  }
}
/*************************************************************/
/*************************************************************
Name:         manual_handler

Called By:    go_handler

DESCRIPTION:  This procedure provides the user interface
necessary to manually change the threshold parameters used
by the program to determine the element durations, the
number of FFT points to use and the peak thresholds used
to determine if a signal is on or off.  The manual_handler allows
the user to use the cursor key to select the parameter to
be changed, then enter the new value.

Globals Changed:  do_manual, and those that are passed by
                        address to the procedure.
*************************************************************/

void manual_handler(int *lowest_intrachar,int *lowest_char,int *lowest_word,
                                                    int *lowest_dit,int *lowest_dah,int *peak_
thresh,
                                                    int *channel)
{
  int orig_x, orig_y, posx, posy, temp=0, key=0;

  orig_x = wherex();
  orig_y = wherey();

  gotoxy(1,1);
  printf("                                    ");

  gettext(WUX,WUY,WLX,WLY,save_pmenu);
  puttext(WUX,WUY,WLX,WLY,param_menu);
  window(WUX,WUY,WLX,WLY);

  gotoxy(25,5);  printf("%4d",*lowest_intrachar);
  gotoxy(25,6);  printf("%4d",*lowest_char);
  gotoxy(25,7);  printf("%4d",*lowest_word);
  gotoxy(25,8);  printf("%4d",*lowest_dit);
  gotoxy(25,9);  printf("%4d",*lowest_dah);
  gotoxy(25,10); printf("%4d",*peak_thresh);
  gotoxy(25,11); printf("%4d",fftpts);
  gotoxy(25,12); printf("%4d",*channel);

  gotoxy(3,5);

  while (do_manual)
          {
```

```
            posy = wherey();
            gotoxy(3,posy);
            key = getch();
            if (key==13)
              {
              if (posy==13)
                        key = 81;
              else
                        {
                        posx = wherex();
                        gotoxy(25,posy);
                        scanf("%4d",&temp);
                        if (posy==5)
                          *lowest_intrachar = temp;
                        else if (posy==6)
                          *lowest_char = temp;
                        else if (posy==7)
                          *lowest_word = temp;
                        else if (posy==8)
                          *lowest_dit = temp;
                        else if (posy==9)
                          *lowest_dah = temp;
                        else if (posy==10)
                          *peak_thresh = temp;
                        else if (posy==11)
                          {
                          fftpts = temp;
                          download();
                          }
                        else if (posy==12)
                          {
                          *channel = temp;
                          download();
                          }
                        gotoxy(25,posy);
                        printf("%4d",temp);
                        gotoxy(posx,posy);
                        }
              }
            if ((key==81)  ||(key==113))
              do_manual = 0;
            if ((key==72))
              {
              putchar(' ');
              if (posy==5)
                        gotoxy(3,13);
              else
                        gotoxy(3,posy-1);
              putchar(16);
              }
            if ((key==80))
              {
              putchar(' ');
              if (posy==13)
                        gotoxy(3,5);
              else
                        gotoxy(3,posy+1);
              putchar(16);
              }
            }
window(1,1,80,25);
```

```
  puttext(WUX,WUY,WLX,WLY,save_pmenu);
  puttext(MUX,MUY,MLX,MLY,go_menu);
  gotoxy(orig_x,orig_y);
}
/*************************************************************/
/*************************************************************
Name:         see_handler

Called By:    This procedure has not been integrated into the
                  present program.  (called by main)

Description:  This procedure allows the user to see the
magnitudes and the durations of samples on a specified channel.
This is a diagnostic tool used to set the operating parameters
manually if the program cannot determine them well enough
itself.
*************************************************************/

void see_handler()
{
  int key=0;

  puttext(SUX,SUY,SLX,SLY,clear_display);
  puttext(MUX,MUY,MLX,MLY,clear_menu);
  gotoxy(60,24);
  printf("Q - Quit");
  gotoxy(35,12);

  printf("See Magnitudes and Durations");
  while (do_see)
            {
            gotoxy(60,25);
            key = getch();
            if ((key==81)||(key==113))
              do_see = 0;
            }
  puttext(SUX,SUY,SLX,SLY,clear_display);
  puttext(MUX,MUY,MLX,MLY,main_menu);
}
/*************************************************************/
/*************************************************************
Name:         go_handler

Called By:    main

Calls:        manual_handler, get_character

DESCRIPTION:  This procedure controls the user interface
for the Go - Menu.  It is entered when the user selects Go
at the main menu and is exited when the user selects Quit.
The user can choose to vary the operating parameters.  Default
operation is to translate Morse code and print it to the
screen.

Globals Changed: None

*************************************************************/

void go_handler()

{
```

```
  int i, posy, posx, choice;

  puttext(SUX,SUY,SLX,SLY,clear_display);
  puttext(MUX,MUY,MLX,MLY,go_menu);

  while (do_go)
            {
            if (do_manual==1)
              {
              gotoxy(1,1);
              printf("Which Channel? ");
              scanf("%d",&choice);
              if (choice == 1)
                        manual_handler(&lowest_intrachar1,&lowest_char1,&lowest_word1,
                                                        &lowest_dit1,&lowest_dah1,&peak_thresh1
,&channel1);
              else
                        manual_handler(&lowest_intrachar2,&lowest_char2,&lowest_word2,
                                                        &lowest_dit2,&lowest_dah2,&peak_thresh2
,&channel2);
              }
            get_character();
            }
  puttext(MUX,MUY,MLX,MLY,main_menu);
}
/*************************************************************/
/********************** Main Program ************************/

main ()

{
int  key;

do_go = 0;
do_see = 0;
do_find = 0;
do_manual = 0;
do_recap = 0;

clrscr();

wpm = 13;
fftpts = 32;


/** Initialize Parameters **/
/** Note that this is done because the auto
    startup is not functioning properly     **/

channel1 = 3;
peak_thresh1 = 1500;
lowest_dah1 = 16;
lowest_dit1 = 1;
lowest_intrachar1 = 1;
lowest_char1 = 17;
lowest_word1 = 41;

channel2 = 11;
peak_thresh2 = 1500;
lowest_dah2 = 12;
lowest_dit2 = 2;
```

```
lowest_intrachar2 = 2;
lowest_char2 = 12;
lowest_word2 = 30;


/**  Initialize Window-Relative Screen Coordinates  **/

xloc1 = 1;
xloc2 = 1;
yloc1 = 1;
yloc2 = 1;

default_addr();
ctrl_break();

init();

dsp_dl_int (flag,1);

setup();

/** Begin execution from the main menu **/
/** if ESC is pressed, loop is terminated **/

while(1)
  {
  if (kbhit())
            {
            key = getch();
            if (key==27)
              exit_routine();
            if ((key==71)||(key==103))
              {
              do_go = 1;
              go_handler();
              }
            if ((key==83)||(key==115))
              {
              do_see = 1;
              see_handler();
              }
            }
  key = 0;
  }

}
```

APPENDIX D: CODE_DSP.C

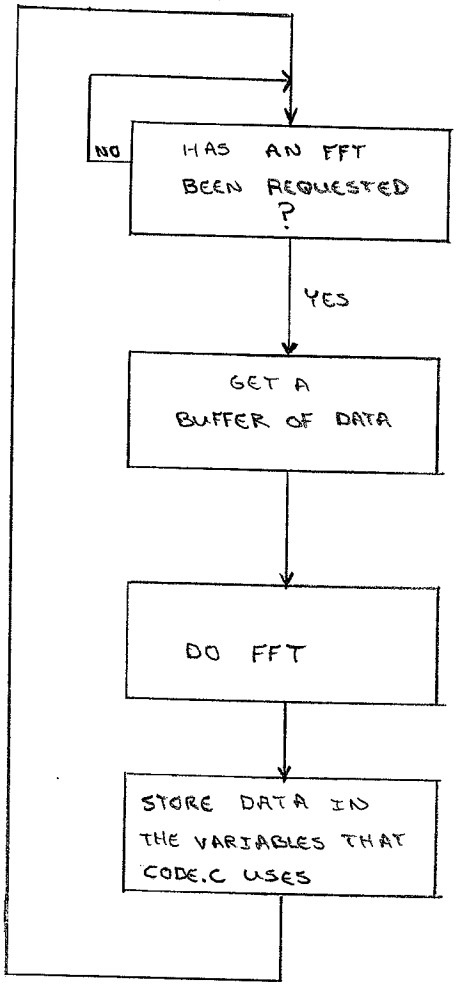


Figure D-3: Flowchart for Code_DSP.C

```
/*******************************************************
filename:    CODE_DSP.C
programmer:  Jeff McDow
revised:     4-29-91
additions:   5-9-91

original programmer:  Kristjan Jonsson
original filename:    CW_DSP.C
date of origin:       5-2-90

Description:  This program is run by the AT&T DSP
board.  When signaled by the program CODE.EXE, which
is being run by the host computer, CODE_DSP.EXE
will perform an FFT on the samples it receives from
the CODEC; store the magnitudes of the specified
bin_numbers--chan1 and chan2; and signal CODE.EXE
to upload these values.

Special Notes:  This program must be compiled using the
AT&T DSP compiler.
*******************************************************/

#include "g:\engrapp\att\dsp32sl\include\io.asm"
#include "g:\engrapp\att\dsp32sl\include\libap.h"

#define MAX 500

/******************** Global Variables ******************/

int buffer[MAX], mag1, mag2, ph, chan1, chan2, n, m;
float buf[MAX];
int flag=0;

/*********************************************************/
/********************** MAIN PROGRAM *********************/
/*********************************************************/
main()
{
set_ioc(0x547);
set_dauc(0);

while(!flag);                   /** Wait for signal from code.exe **/

m = (int) log2((float) n);      /** When signal is received,   **/
                                        /** get ready to do an FFT     **/

while(1)                        /** Main Loop **/
{  while (!flag);               /** Wait for   signal from code.exe **/
   get_data();                 /** Collect a buffer of data from the CODEC **/
   fft(n,m,buf);                    /** do an n point FFT on data in buf **/

   ritomp();                                    /** convert from real-imaginary pairs,**/
                                                 /** to magnitudes                      **/

   mag1 = buffer[chan1];        /** store magnitude of chan1  **/
   mag2 = buffer[chan2];
   flag = 0;                    /** signal code.exe that FFT is done **/
   }
}

/*********************************************************
                        FUNCTION DEFINITIONS
```

```
**********************************************************/
/**********************************************************
Name:         get_data

Called By:    main

Description:  This function gets a buffer of data from the
CODEC and stores it in buf.
**********************************************************/

get_data ()

{
  register int i;
  register float *p;

  p = buf;
  i = n;

  while (i--)
   {
   *p++ = ic_ibuf();
   *p++ = 0;
   }
}
/**********************************************************
**********************************************************
Name:         ritomp

Called By:    main

Description:  This function converts the data in buf
(after an FFT has been performed on it) from real-imaginary
pairs into a series of magnitudes.
**********************************************************/

ritomp ()

{ register int i, *p2;
  register float *p1, real, imag;

  p1 = buf;
  p2 = buffer;
  i = n;

  while (i--)
           {
           real = *p1++;
           imag = *p1++;
           *p2++ = (int) sqrt(real*real + imag*imag);
           }
}
/**********************************************************/
```

APPENDIX E

SYSTEM SPECIFICATIONS

APPENDIX E: SYSTEM SPECIFICATIONS: PHASE-LOCK LOOP SYSTEM

HARDWARE

Description of Operation: See pages 2, 3, 7, 8, 29

Schematic: See pages 30, 31

Components Used:

Integrated Circuits

Part	Quantity
LS244 Octal Buffer	2
LS245 Octal Bus Transceiver	1
LS85 4-Bit Comparator	1
LS138 3 to 8 Decoder	1
LS00 Quad 2-Input NAND	1
LS04 Hex Inverters	1
LS08 Quad 2-Input AND	1
LS541 Octal Buffer	1
LS155 Dual 2 to 4 Decoder	2
LS373 Octal D Latch	1
LM567 Tone Decoder	1

Other Parts

Part	Quantity
JDR PR-2 Interface Card	1
4.7 K-Ohm Resistor	1
100 Ohm Resistor	1
50 K-Ohm Potentiometer	1
0.01 uF Ceramic Capacitor	1
0.05 uF Ceramic Capacitor	2
10 uF Tantalum Capacitor (15-25 V)	3
0.1 uF Tantalum Capacitor	8
2.2 uF Tantalum Capacitor	1
Jumbo Green LED	1
Speaker Jack	1

Note: System requires an IBM-compatible personal computer.

SOFTWARE

Name of Executable File: MORSE.EXE

Size of Executable File: 31503 bytes

Listing of Source File: See pages 45 - 52

PERFORMANCE

Maximum Translation Speed: 40 WPM on an 8088 IBM PC

SPECIFICATIONS: AT&T DSP SYSTEM

HARDWARE

AT&T DSP32 Floating Point Digital Signal Processor

8-bit CODEC Sampling at 8 kHz for Data Acquisition

SOFTWARE

DSP-Based Program: CODE_DSP.C

Size of Executable File: 10596 bytes

Listing of Source File: See pages 72-73

PC-Based Program: CODE.C

Size of Executable File: 88470 bytes

Listing of Source File: See pages 54 - 70

PERFORMANCE

Maximum Translation Speed: 70 WPM with 75% Accuracy

60 WPM with 100% Accuracy

Note: Speed measurements were made on a 80386 PC

APPENDIX F

FUTURE ADDITIONS