Dynamics Technology, Inc.

DTW-8944-91003

METRIWAVE FINAL REPORT NAS7-1071

[illegible]

[illegible] 226

P. 223

By:

Wyman Williams

OCTOBER 1991

Dynamics Technology, Inc.
21311 Hawthorne Boulevard
Suite 300
Torrance, CA 90503
(310) 543-5433

508964

TABLE OF CONTENTS

LIST OF FIGURES

1.0 INTRODUCTION

The superconductor-insulator-superconductor (SIS) mixer is a device which is being used in the construction of very sensitive receivers in the millimeter and submillimeter wavelength regions. With its potential for conversion gain and quantum-limited performance, it is becoming a device of prime importance in radio astronomy as well as earth and planetary atmospheric research.

Many of the parameters of the SIS mixer cannot be readily measured in the laboratory, however, since most commercially available test instruments use test signal powers large enough to saturate or destroy SIS junctions. This report details the construction of a microwave network analyzer with extremely low test signal powers. It documents the results of a development performed by Dynamics Technology, Inc., under a Phase II SBIR contract from NASA (NAS7-1025).

The work resulted in a network analyzer to be delivered to workers at the Jet Propulsion Laboratory, which should be capable of SIS mixer characterization in support of their ongoing work in this area.

This report is organized in sections which correspond to the technical objectives identified in the Phase II proposal. Appendices provide detailed schematics and parts lists of all circuitry designed in this work. Circuit descriptions in the main text of the report support the schematics. In addition, another appendix contains an operator's manual for the delivered network analyzer. These Phase II technical objectives were:

1. Develop parametric variations of the existing design of the sampled-line reflectometer modules. Evaluate the performance of the variations. Build and test modules with desirable performance parameters.
2. Design instrumentation preamplifiers and A/D converters for use with the network analyzer. Build and test the preamplifiers.
3. Design and build synchronization circuitry for use with the network analyzer.
4. Refine the existing phase shifter design.
5. Design and build a microwave signal generator for use with the network analyzer.
6. Deliver a working prototype network analyzer to NASA.

The sections following this introduction present theoretical and experimental results from work on each of the above objectives.

2.0 SAMPLED-LINE MODULE DESIGN

For the sake of completeness, the theoretical background for the sampled-line network analyzer is presented here. Some new theoretical results derived as part of the Phase ii effort are presented. This is followed by a description of the design and testing of the new sampled-line modules.

2.1 Four-Port Network Analyzer Theory

The job a network analyzer must perform can be stated quite simply. The parameters are shown in Figure 2.1. Two travelling voltage waves, a and b, propagate in opposite directions on a transmission line. At a particular point on the line, known as the reference plane, the network analyzer measures the amplitude ratio and phase difference between these two (sinusoidal) waves.

The ability to measure this complex ratio, for a set of transmission lines connected to the ports of an unknown microwave network, allows the network analyzer to determine the complete S-parameters of that network. The machinations required to derive all the S-parameters from a given set of measurements, however, become more elaborate as the number of ports on the unknown network increases.

Machines that measure the complex ratio of travelling waves fall into two categories. The first type is built by combining a linear four-port network with a vector voltmeter, and the second uses a six-port network and several power detectors.

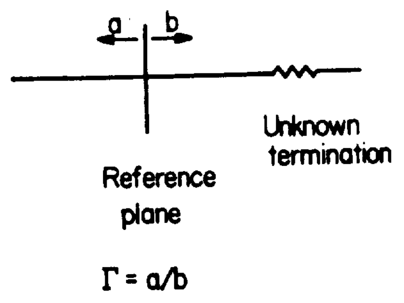


Figure 2.1 The normalized scattering waves a and b are related to the forward-going and reverse-going voltage waves V^+ and V^- on the transmission line by $a = V^+/\sqrt{2Z_0}$ and $b = V^-/\sqrt{2Z_0}$ where Z_0 is the characteristic impedance of the transmission line. Thus, Γ is the voltage reflection coefficient, $\Gamma = V^+/V^-$.

The four-port network analyzer was the first developed, and forms the basis of most of the currently available commercial network analyzers. In this chapter, the theory of operation of the four-port network analyzer will be presented. The theories of operation of the six-port and sampled-line network analyzers build upon this theory, and assumptions made in developing the four-port theory are germane to comparisons of performance of the various network analyzer approaches.

2.1.1 Reflection Measurements with the Four-Port Network Analyzer

The reflection measurement is the simplest network analyzer measurement. As shown in Figure 2.1, only an unknown impedance and two travelling wave quantities are involved. The unknown impedance is connected to the end of a transmission line. The voltage wave a, travelling on the line impinges on the unknown device. The travelling wave b, resulting from the impedance mismatch between the unknown device and the transmission line, is reflected back along the line. By measuring the amplitude ratio and phase difference between the incident and reflected waves at a fixed reference plane, the complex reflection coefficient, Γ of the unknown device is determined. This, along with knowledge of the characteristic impedance of the transmission line, is sufficient to determine the unknown impedance.

Figure 2.2 shows schematically how a vector voltmeter and a pair of matched directional couplers can be combined to yield a four-port reflectometer (the box containing the two couplers forms the "four-port" for which the analyzer is named). The vector voltmeter is a frequency-tracking heterodyne receiver that indicates the complex ratio of the two microwave signals presented at its inputs.

One directional coupler samples the forward-going wave, and the other the reflected wave. If the directional couplers are matched and have perfect directivity, and if port 2 presents a source impedance of exactly Z_0 to the device under test, then $b_3/b_4 = a_2/b_2 = \Gamma$ and the value indicated by the vector voltmeter will be the true reflection coefficient.

These conditions are stringent, however, and even with the best designs imperfections in the components give rise to large errors. With the Hewlett-Packard 8743B reflection-transmission test unit, for example, in the 2-8 GHz frequency range, magnitude errors of 0.15 are possible when measuring reflection coefficients of unity magnitude.

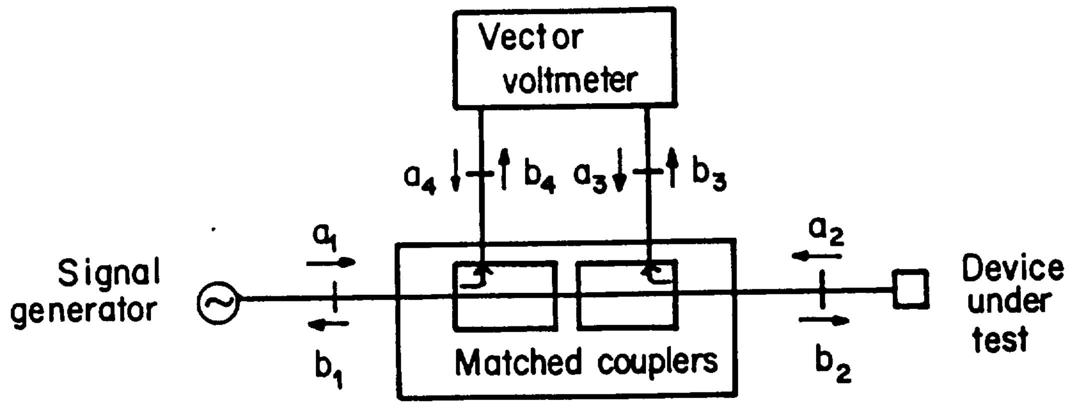


Figure 2.2 The four-port network analyzer uses a vector voltmeter and a pair of matched directional couplers to measure the reflection coefficient.

These large errors, however, are systematic and can be calibrated out by measuring known calibration standards. The calibration procedure and the number of standards required are determined by the relationship between the true and indicated values of Γ. This relationship can be found in a straightforward way by adapting Middlebrook's extra element theorem[1] to travelling wave network parameters.

Consider the four-port network analyzer of Figure 2.3. Here the dual directional couplers of Figure 2.2 have been replaced by an arbitrary linear four-port. Also, offset and gain terms have been added to the response of the vector voltmeter.

By superposition,

$$b_4 = K_1 a_1 + K_2 a_2 \tag{2.1}$$

$$b_3 = L_1 a_1 + L_2 a_2 \tag{2.2}$$

$$b_2 = M_1 a_1 + M_2 a_2 \tag{2.3}$$

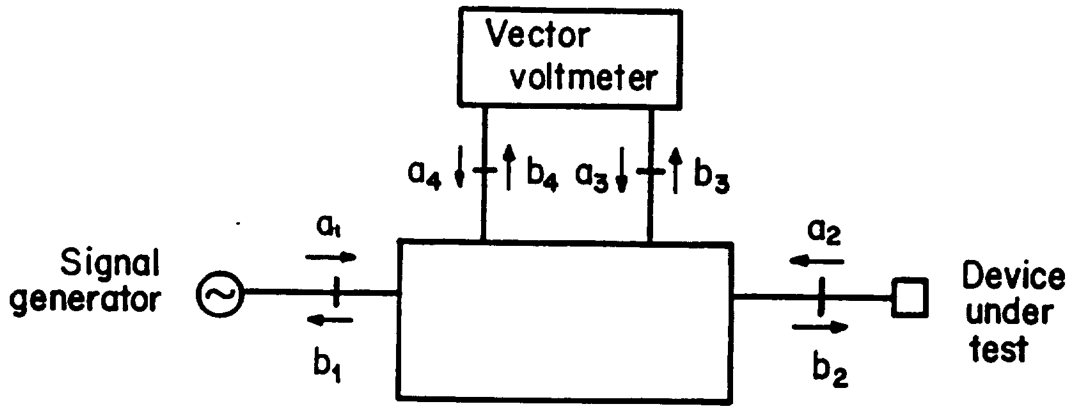


Figure 2.3 This four-port network analyzer uses a vector voltmeter and an arbitrary linear four-port network to measure the reflection coefficient. If b_3/b_4 exists, it is just a bilinear transform of Γ.

By definition, $a_2 = \Gamma b_2$. Inserting this in (2.1 - 2.3) and solving (2.3) for b_2 gives

$$b_4 = K_1 a_1 + K_2 \Gamma b_2 \tag{2.4}$$

$$b_3 = L_1 a_1 + L_2 \Gamma b_2 \tag{2.5}$$

$$b_2 = \frac{M_1}{1 - M_2 \Gamma} a_1 \tag{2.6}$$

Substituting (2.6) into (2.4) and (2.5) gives

$$b_4 = \frac{K_1 + \left(M_1 K_2 - M_2 K_1\right)\Gamma}{1 - M_2 \Gamma} a_1 \tag{2.7}$$

$$b_3 = \frac{L_1 + \left(M_1 L_2 - M_2 L_1\right)\Gamma}{1 - M_2 \Gamma} a_1 \tag{2.8}$$

Dividing (2.8) by (2.7) gives

$$\frac{b_3}{b_4} = \frac{L_1 + \left(M_1L_2 - M_2L_1\right)\Gamma}{K_1 + \left(M_1K_2 - M_2K_1\right)\Gamma} \tag{2.9}$$

Finally, including the response of the vector voltmeter, $\Gamma' = C_0 + C_1\, b_3/b_4$ and dividing the numerator and denominator through by the coefficient of Γ in the numerator gives

$$\Gamma' = \frac{\Gamma + A}{B\Gamma + C} \tag{2.10}$$

So the final result is that the value Γ' indicated on the vector voltmeter is a bilinear transform of the actual value of Γ. The bilinear transform has three complex constants, so three known standards must be measured in order to determine the calibration coefficients, and thereafter correct the measured data.

The calibration procedure is straightforward. Multiplying (2.10) through by the denominator of the right side gives

$$-A + \Gamma\Gamma' B + \Gamma' C = \Gamma \tag{2.11}$$

which is a linear equation in A, B, and C. By observing Γ' for three different known values of Γ, the coefficients of three such equations are determined, and the values of A, B, and C can be determined. The true value of Γ is then found by inverting the transform:

$$\Gamma = \frac{C\Gamma' - A}{1 - B\Gamma'} \tag{2.12}$$

The above result is interesting in several respects. The first is its generality. Any four-port network that is linear, and for which the ratio of (2.10) exists, can be used as a network analyzer and calibrated by measuring three known standards. Also, the fact that Γ' is a ratio measurement makes the result independent of the level of the test signal injected at port 1 and thus the source impedance of the generator.

The fact that Γ' is a bilinear transform of Γ allows the systematic errors of the system to be modeled in a simple way. Figure 2.4 shows the error model. It consists of an ideal network analyzer with an "error two-port" between the ideal analyzer's reference plane and the device under test. The fact that this gives rise to the same relationship between observed and actual values as that derived above can be seen by considering the T-parameters of the error two-port.

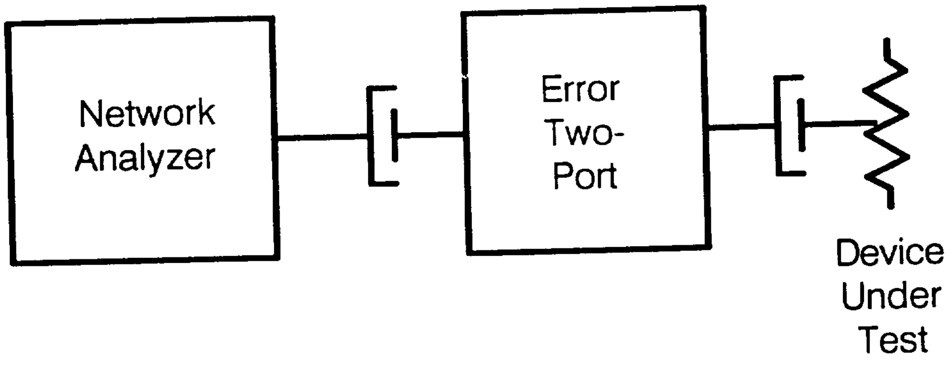


Figure 2.4 Errors in the reflection measurement can be modeled by an error 2-port between the reflectometer and the device under test.

$$b_1 = T_{11}a_2 + T_{12}b_2 \tag{2.13}$$

$$a_1 = T_{21}a_2 + T_{22}b_2 \tag{2.14}$$

$$\Gamma' = \frac{b_1}{a_1} = \frac{T_{11}a_2 + T_{12}b_2}{T_{21}a_2 + T_{22}b_2} \tag{2.15}$$

$$= \frac{T_{11}\Gamma + T_{12}}{T_{21}\Gamma + T_{22}} \tag{2.16}$$

For a practical system $T_{11} \neq 0$ and dividing the numerator and denominator of (2.16) through by T_{11} gives (2.10).

This error model can be represented by the signal flow graph of Figure 2.5. The graph is labeled with three error terms that are often quoted for reflectometers. They are related to the physical sources of error in the dual-coupler type reflection test set. E_D is the directivity error, related to the imperfect directivity of the couplers in the test set. E_S is the source match error, related to the impedance mismatch at the measurement port of the test set. Lastly, E_R is the frequency response error, related to differences in the frequency responses of the two directional couplers.

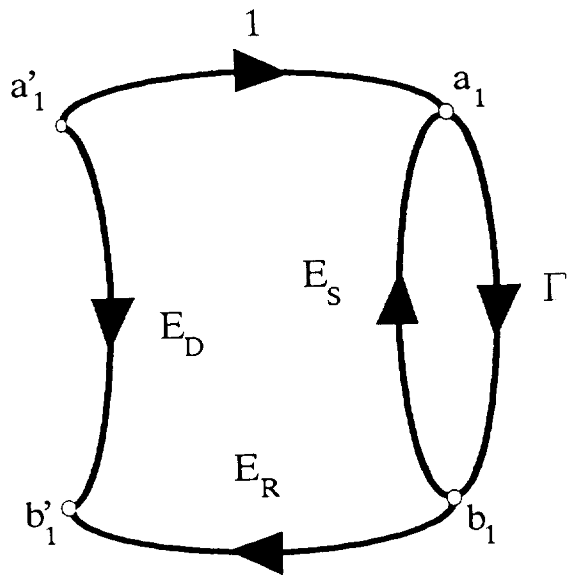


Figure 2.5 Signal flow graph for errors in the reflection measurement.

In terms of these quantities, the relation between Γ and Γ' can be found through Mason's gain rule as

$$\Gamma' = E_D + \frac{E_R \Gamma}{1 - E_S \Gamma} \tag{2.17}$$

$$= \frac{E_D + (E_R - E_S E_D)\Gamma}{1 - E_S \Gamma} \tag{2.18}$$

In practical systems, the greatest errors typically arise from the source mismatch error, when Γ is near unity in magnitude.

In modern network analyzers, built-in computers store the calibration constants and perform the bilinear transform to display the corrected reflection coefficient. Specifications are still often quoted, however, in terms of effective directivity, source match, etc. This method of specification makes sense due to a property of the bilinear transform: if Γ' is a bilinear transform of Γ, and Γ'' is a bilinear transform of Γ', then Γ'' is a bilinear transform of Γ. Thus, if there are errors in determination of the calibration coefficients above, and the transform is applied to the measured data, one may still be assured that the result is a bilinear transform of the true Γ, and this bilinear transform can be classified in terms of the quantities on a signal flow graph like that of Figure 2.5.

If it is assumed that perfect calibration standards are used, and the terms of the bilinear transform are found without error, then the only sources of error remaining in the system are those due to noise and nonlinearities in the vector voltmeter.

In the vector voltmeter, the test and reference microwave signals are down-converted to intermediate frequencies of a few MHz or hundreds of kHz by a sampling mixer. At the i-f, the test signal is split into two components, one in phase with the reference signal and one in quadrature with it. From these two components, the real and imaginary parts of the complex ratio are derived. If the gains of the two channels for the test signal are not precisely matched, or if the quadrature signal is not truly orthogonal to the in-phase signal, then errors will result which cannot be calibrated out by the linear procedures described above.

Since the i-f circuitry needs only to be adjusted for operation at a single frequency, however, sufficient precision can be achieved. In modern computer-corrected network analyzers, the errors due to these nonlinearities are less than the errors resulting from lack of repeatability of the microwave connectors used to connect the device under test to the network analyzer. This lack of repeatability - the inability to exactly match all the connectors on all the calibration standards - is the factor limiting accuracy of reflection measurements with these analyzers.

2.1.2 S-Parameter Measurements with the Four-Port Network Analyzer

The situation is a bit more complicated when the S-parameters of an unknown two-port are to be measured. Transmission as well as reflection must be measured. A setup for making this measurement is shown in Figure 2.6.

The test set contains a reflectometer like that described above. Port 1 of the device under test is connected to this reflectometer test port. Another port, the "transmission return" receives the signal emerging from port 2. A coaxial relay selects which signal is presented to the vector voltmeter for the complex ratio measurement. To perform a full S-parameter measurement, the unknown two-port must be flipped end-for-end once during the measurement, and the coaxial relay switched each time to observe the reflection at both the network's ports, and the transmission through it in both directions.

This gives a total of four measurements. Expressions will be derived here for the values indicated by the vector voltmeter for each of these measurements in terms of the S-parameters of the device under test. These will then be inverted to yield expressions for these S-parameters in terms of the measured quantities.

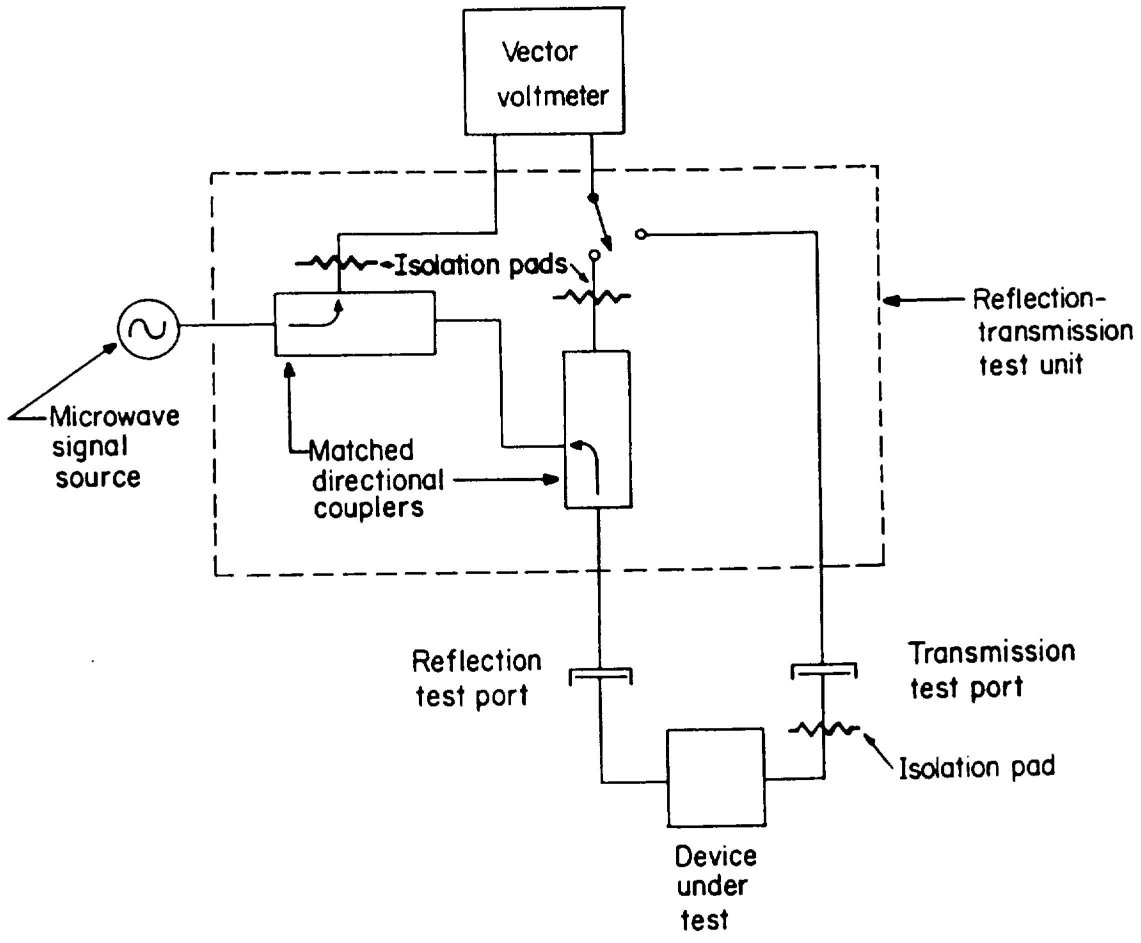


Figure 2.6 A reflection-transmission test set for full S-parameter measurements using a 4-port network analyzer.

The reflection measurement is the same as that described in the previous section, except that here the measurement is of the reflection coefficient of port 1 of the two-port with its port 2 terminated in Γ_L, the impedance of the transmission return. This reflection coefficient, Γ_1 is given by

$$\Gamma_1 = S_{11} + \frac{S_{12}S_{21}\Gamma_L}{1 - S_{22}\Gamma_L} \tag{2.19}$$

where the S-parameters are those of the 2-port that is being measured. Substituting this into equation (2.10) yields

$$S'_{11} = \frac{\left(S_{11} + \dfrac{S_{12}S_{21}\Gamma_L}{1 - S_{22}\Gamma_L}\right) + A}{B\left(S_{11} + \dfrac{S_{12}S_{21}\Gamma_L}{1 - S_{22}\Gamma_L}\right) + C} \tag{2.20}$$

$$= \frac{\left(S_{11} - \Gamma_L\Delta\right) + A\left(1 - S_{22}\Gamma_L\right)}{B\left(S_{11} - \Gamma_L\Delta\right) + C\left(1 - S_{22}\Gamma_L\right)} \tag{2.21}$$

where S'_{11} is the value indicated by the vector voltmeter and $\Delta \equiv S_{11}\, S_{22} - S_{12}\, S_{21}$ is the system determinant of the two-port being measured.

When performing the transmission measurement, port 2 of the device under test is connected directly to the vector voltmeter, so the meter measures b_5/b_4. To derive an expression for b_5/b_4, equation (2.6) is first rearranged to give

$$a_1 = \frac{1 - M_{2T}\,\Gamma}{M_{1T}}\, b_2 \tag{2.22}$$

The subscript "T" is added to the coefficients of this expression because, in general, when the coaxial switch is moved from the reflection to the transmission position, the values of the K's, L's and M's in equations (2.1 - 2.8) will change, due to a change in the reflection coefficient seen looking out port 3 of the four-port network. The value of Γ_L, the reflection coefficient of the transmission return, will also change in general when this switch is moved, to a value of Γ_{LT}.

This noted, the derivation of b_5/b_4 continues by substituting (2.22) into (2.4) to give

$$b_4 = \frac{1}{M_{1T}} [K_{1T} + (M_{1T} K_{2T} - M_{2T} K_{1T})\Gamma] b_2 \tag{2.23}$$

$$= P(B_T\Gamma + C_T) b_2 \tag{2.24}$$

where B_T and C_T have the same functional form as B and C in equation (2.10), but with T subscripts on the L's and M's.

The value of Γ in (2.22 - 2.24) above is the reflection coefficient looking into port 1 of the two-port being measured when its second port sees Γ_{LT}. Using (2.19), the expression for b_4 can be expanded to give

$$b_4 = P\left[B_T\left(S_{11} + \frac{S_{12}S_{21}\Gamma_{LT}}{1 - S_{22}\Gamma_{LT}}\right) + C_T\right] b_2 \tag{2.25}$$

An expression for b_5 can be written in terms of b_2, as well:

$$b_5 = \frac{NS_{21}}{1 - S_{22}\Gamma_{LT}} b_2 \tag{2.26}$$

where N is the gain of the path from the transmission return port to the vector voltmeter's port.

Dividing (2.26) by (2.25) gives

$$\frac{b_5}{b_4} = \frac{1}{P} \frac{\dfrac{NS_{21}}{1 - S_{22}\Gamma_{LT}}}{B_T\left(S_{11} + \dfrac{S_{12}S_{21}\Gamma_{LT}}{1 - S_{22}\Gamma_{LT}}\right) + C_T} \tag{2.27}$$

$$= \frac{1}{P} \frac{NS_{21}}{B_T(S_{11} - \Gamma_{LT}\Delta) + C_T\,(1 - S_{22}\Gamma_{LT})} \tag{2.28}$$

Applying the offset and gain terms of the vector voltmeter ($S'_{21} = C_0 + C_1\, b_5/b_4$) gives

$$S'_{21} = D + \frac{ES_{21}}{B_T\left(S_{11} - \Gamma_{LT}\Delta\right) + C_T\left(1 - S_{22}\Gamma_{LT}\right)} \tag{2.29}$$

where S'_{21} is the value indicated by the vector voltmeter when the coaxial switch is in the transmission measurement position. Equations (2.21) and (2.29) are two equations in the unknown S-parameters resulting from the measurements. Flipping the two-port under test end for end and measuring it in the reverse direction yields two more equations.

$$S'_{22} = \frac{\left(S_{22} - \Gamma_L\Delta\right) + A\left(1 - S_{11}\Gamma_L\right)}{B\left(S_{22} - \Gamma_L\Delta\right) + C\left(1 - S_{11}\Gamma_L\right)} \tag{2.30}$$

$$S'_{12} = D + \frac{ES_{12}}{B_T\left(S_{22} - \Gamma_{LT}\Delta\right) + C_T\left(1 - S_{11}\Gamma_{LT}\right)} \tag{2.31}$$

Equations (2.21) and (2.29 - 2.31) give four equations in the four unknown S-parameters. Unfortunately, these equations are nonlinear, and cannot be inverted in closed form to give the S-parameters[2]. This difficulty is removed, however, if Γ_L, B and C do not change when the vector voltmeter is switched from reflection to transmission measurement. Taking $\Gamma_L = \Gamma_{LT}$, $B_T = B$ and $C_T = C$ the above equations can be solved. This inversion yields expressions for the true S-parameters in terms of the measured values:

$$S_{11} = \frac{E^2\left(A - CS'_{11}\right)\left(BS'_{22} - 1\right) - \Gamma_L\left(AB - C\right)^2\left(S'_{12} - D\right)\left(S'_{21} - D\right)}{E^2\left(BS'_{11} - 1\right)\left(BS'_{22} - 1\right) - \Gamma_L^2\left(AB - C\right)^2\left(S'_{12} - D\right)\left(S'_{21} - D\right)} \tag{2.32}$$

$$S_{12} = \frac{E\left(AB - C\right)\left(S'_{12} - D\right)\left[\left(BS'_{11} - 1\right) - \Gamma_L\left(A - CS'_{11}\right)\right]}{E^2\left(BS'_{11} - 1\right)\left(BS'_{22} - 1\right) - \Gamma_L^2\left(AB - C\right)^2\left(S'_{12} - D\right)\left(S'_{21} - D\right)} \tag{2.33}$$

$$S_{21} = \frac{E\left(AB - C\right)\left(S'_{21} - D\right)\left[\left(BS'_{22} - 1\right) - \Gamma_L\left(A - CS'_{22}\right)\right]}{E^2\left(BS'_{11} - 1\right)\left(BS'_{22} - 1\right) - \Gamma_L^2\left(AB - C\right)^2\left(S'_{12} - D\right)\left(S'_{21} - D\right)} \tag{2.34}$$

$$S_{22} = \frac{E^2\left(A - CS'_{22}\right)\left(BS'_{11} - 1\right) - \Gamma_L\left(AB - C\right)^2\left(S'_{12} - D\right)\left(S'_{21} - D\right)}{E^2\left(BS'_{11} - 1\right)\left(BS'_{22} - 1\right) - \Gamma_L^2\left(AB - C\right)^2\left(S'_{12} - D\right)\left(S'_{21} - D\right)} \tag{2.35}$$

In order to use these closed-form solutions and avoid an iterative calculation at each frequency point of the measurement, the hardware of the system is engineered to isolate the detector switch from the RF measurement ports of the analyzer. Attenuators are placed between the four-port network and the vector voltmeter, and between port 2 of the device under test and the transmission return port, as indicated in Figure 2.6. Also, the input port of the vector voltmeter is matched to the line as well as possible, and switches that terminate unselected lines in 50 Ω are used.

With the assumption of switch-independence, then, only six complex constants, A, B, C, D, E, and Γ_L are required to model the linear systematic errors of the reflection-transmission test set. The error model for the test set can again be drawn as a signal flow graph, as shown in Figure 2.7. The three error terms noted above have not changed, but three new ones have been added. E_L is the load mismatch term, related to the mismatch of the transmission return port of the analyzer. E_T is the gain of the transmission return path. E_X is related to the vector voltmeter's offset term. The results above can be rewritten in terms of this error model to give

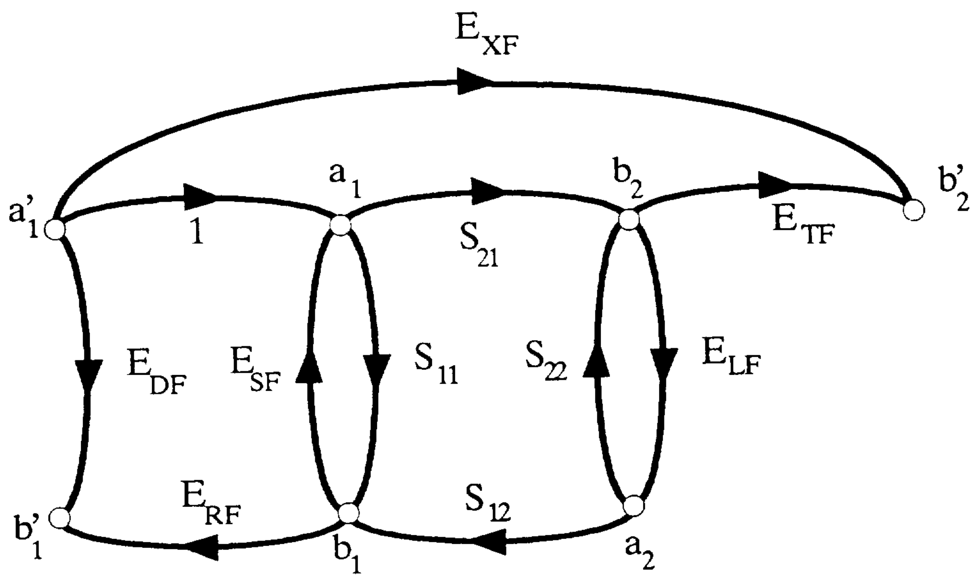


Figure 2.7 This flow graph shows the error model for the reflection-transmission test set.

$$S_{11} = \frac{\left\{\left(\frac{S'_{11} - E_D}{E_R}\right)\left[1 + \left(\frac{S'_{22} - E_D}{E_R}\right)E_S\right]\right\} - \left[\left(\frac{S'_{21} - E_X}{E_T}\right)\left(\frac{S'_{12} - E_X}{E_T}\right)E_L\right]}{\left[1 + \left(\frac{S'_{11} - E_D}{E_R}\right)E_S\right]\left[1 + \left(\frac{S'_{22} - E_D}{E_R}\right)E_S\right] - \left[\left(\frac{S'_{21} - E_X}{E_T}\right)\left(\frac{S'_{12} - E_X}{E_T}\right)E_L^2\right]} \qquad (2.36)$$

$$S_{21} = \frac{\left[1 + \left(\frac{S'_{22} - E_D}{E_R}\right)(E_S - E_L)\right]\left(\frac{S'_{21} - E_X}{E_T}\right)}{\left[1 + \left(\frac{S'_{11} - E_D}{E_R}\right)E_S\right]\left[1 + \left(\frac{S'_{22} - E_D}{E_R}\right)E_S\right] - \left[\left(\frac{S'_{21} - E_X}{E_T}\right)\left(\frac{S'_{12} - E_X}{E_T}\right)E_L^2\right]} \qquad (2.37)$$

with similar results for S_{12} and S_{22}.

So six complex constants (at each frequency of interest) are sufficient to completely characterize the errors of this type of network analyzer. These constants are found by measuring six known standards.

In addition to the assumption of switch-independence, another assumption has been made to arrive at the result above. It is assumed that the only coupling between the reflection and transmission test ports is through the network under test: there are no RF leakage paths (the "leakage path" that gives rise to the D term in (2.29) is in fact just a result of the offset term in the vector voltmeter's response). The assumption of no RF leakage is good in most systems, since coaxial relays, which have very high isolation, are typically used to switch the signal paths. If significant leakage paths do exist, the expressions above are invalid and the closed-form solution impossible.

Finally, for the sake of thoroughness, Figure 2.8 shows the architecture of a full S-parameter test set. This is the configuration used in such state-of-the-art network analyzers as the Hewlett-Packard 8510. It consists essentially of two transmission-reflection test sets of the type just discussed placed back to back. Another coaxial switch selects which end of the setup receives the test signal. Thus, there is no need to flip the device under test end for end. This improves measurement speed and accuracy. As long as the detector switch is isolated from the network as described above, an analysis similar to that above can be performed to yield closed-form expressions for the S-parameters. The analysis is essentially performed twice, once for each position of the source switch. The result is two signal flow graphs like that of Figure 2.7, and an error model which contains twelve complex constants. This is the "full twelve term error model" which is often mentioned in connection with these analyzers.

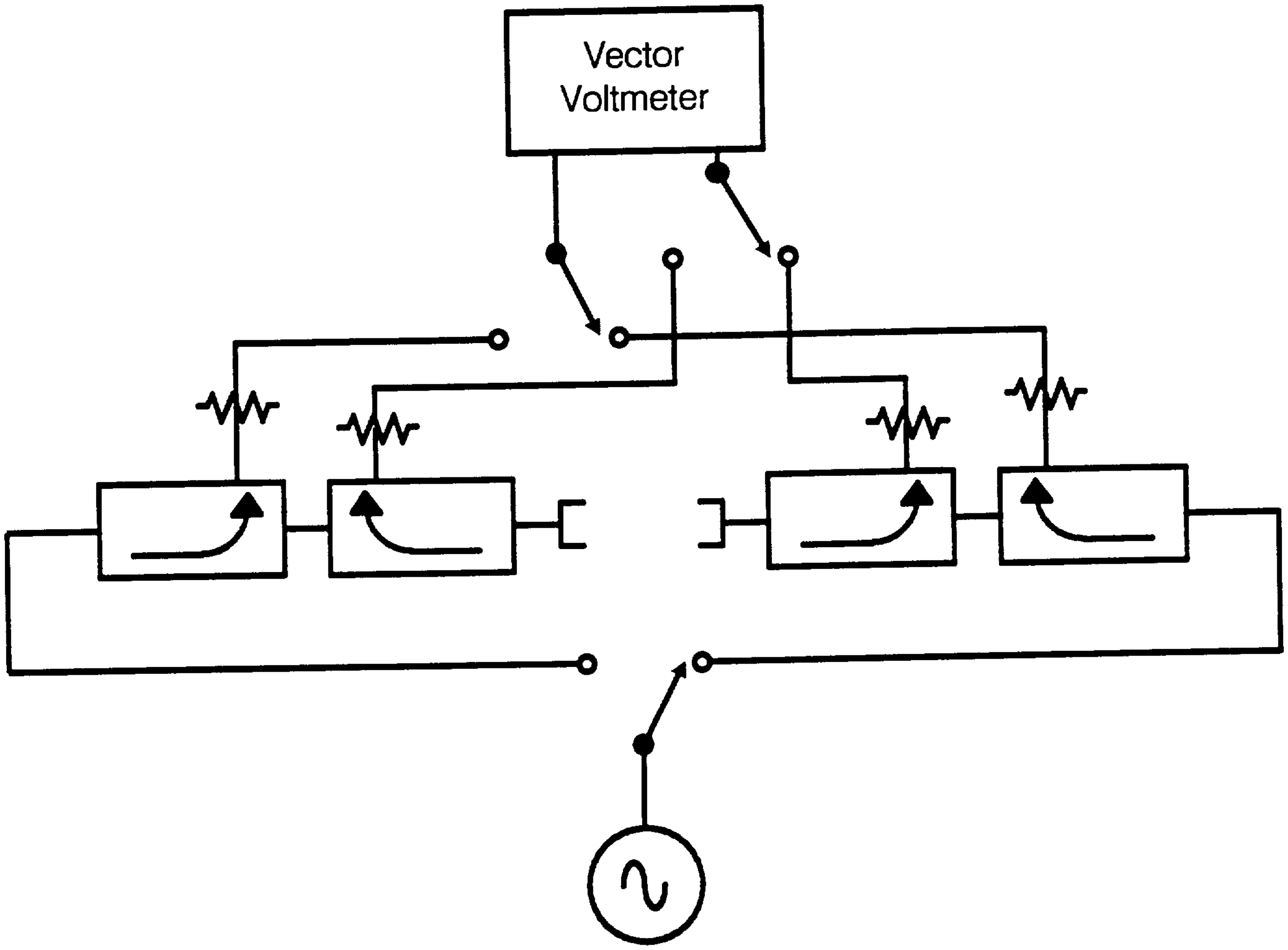


Figure 2.8 A full S-parameter test set using the 4-port network analyzer.

So in summary, four-port type network analyzers measure the S-parameters of two-ports by using a single vector voltmeter with an arrangement of switches and directional couplers to route the appropriate signals to the voltmeter. As long as the networks used satisfy requirements of switch independence and isolation, the S-parameters of the unknown device can be found from the measured data.

As will be seen in the next section, things are simplified somewhat if two vector voltmeters are included in the system, and the number of switches reduced. This is the approach taken when six-port network analyzers are used to measure S-parameters.

2.2 Six-Port Network Analyzer Theory

It was shown in the previous section that if the complex ratio, b_3/b_4, of two voltage waves emerging from a four-port network can be determined, then the reflection coefficient Γ of the unknown load connected to the four-port's test port can be calculated. The six-port network analyzer allows b_3/b_4 to be determined without the use of a vector voltmeter. This reduces the cost and complexity of the system.

Figure 2.9 shows the general six-port network analyzer configuration[3]. Only microwave power detectors are used in this instrument. These are square law detectors; that is $P_i \alpha |b_i|^2$, i = 3..6. By measuring the magnitudes of the voltage waves emerging from port 3, 4, 5 and 6, and doing some trigonometry, the complex ratio of the wave emerging from port 3 to that at port 4 can be calculated. This reduces the six-port to an equivalent four-port network analyzer, which may be calibrated as described in Chapter 2.

The equivalence of the six-port and the four-port analyzers can be seen by noting that the addition of ports 5 and 6 does not change any of the arguments presented in the previous section. One can think of ports 5 and 6, and their terminating impedances being absorbed into the network, leaving a four-port network.

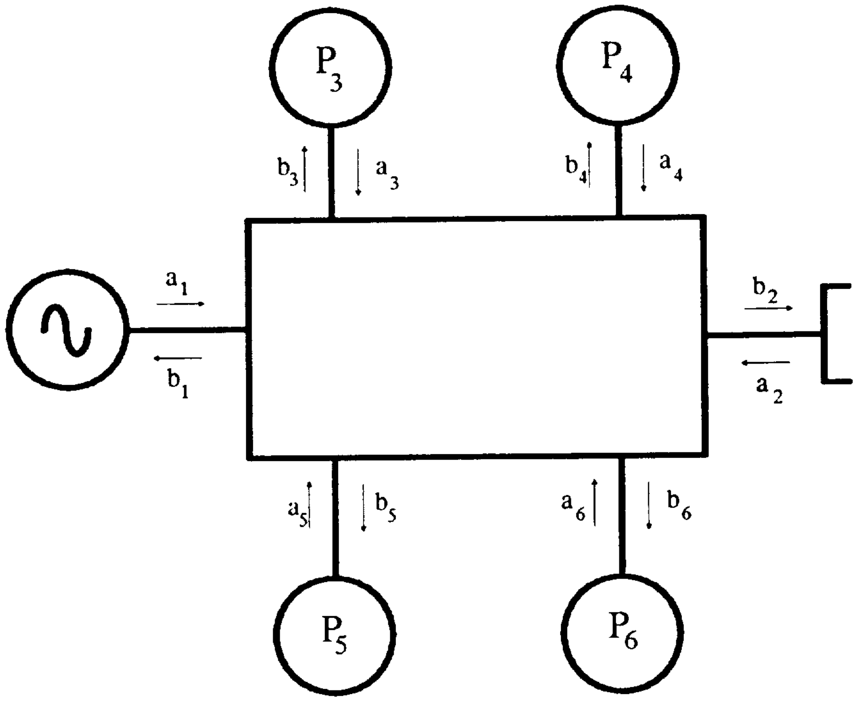


Figure 2.9 The six-port network analyzer uses only power detectors to determine the complex ratio a_2/b_2 at its test port.

The six-port network, with power detectors installed, then, is like a four-port network with a built-in vector voltmeter. Since power detectors are relatively inexpensive, it is usually unnecessary to switch-multiplex them between two six-port networks as was done with the vector voltmeter in the four-port measurement systems described above. The self-contained reflectometer module becomes the basic building block of measurement systems using the six-port network analyzer. As will be seen below, this facilitates measurement architectures and procedures different from those used with the four-port system.

As in Section 2.1, the theory of the reflectometer will be described first, followed by that of the S-parameter measurement system.

2.2.1 Reflection Measurements with the 6-Port Network Analyzer

With the reflectometer of Figure 2.9, the goal is to find the ratio of b_3/b_4. This quantity is defined as w. Once the complex ratio w is found, the same calibration and measurement procedures as were used for the four-port network analyzer can be applied. For this reason, finding w from the six-port's power detector outputs is termed a six-port to four-port conversion. Although w is a bilinear transform of the desired quantity Γ, the complex w-plane is the most convenient in which to work for many of the calculations related to calibration and measurement with the six-port.

The key observation for making the six-port to four-port conversion is that the voltage waves emerging from ports 5 and 6 of the six-port are related to those from ports 3 and 4 in a simple way [4]:

$$b_5 = Kb_3 + Lb_4 \tag{2.38}$$

$$b_6 = Mb_3 + Nb_4 \tag{2.39}$$

Rearranging and taking magnitudes gives

$$\frac{1}{|K|}\left|\frac{b_5}{b_4}\right| = \left|\frac{b_3}{b_4} + \frac{L}{K}\right| \tag{2.40}$$

$$\frac{1}{|M|}\left|\frac{b_6}{b_4}\right| = \left|\frac{b_3}{b_4} + \frac{N}{M}\right| \tag{2.41}$$

Taking $w \equiv b_3/b_4$, these expressions can be written as

$$|w| = \left|\frac{b_3}{b_4}\right| \tag{2.42}$$

$$|w - w_1| = \frac{1}{|K|}\left|\frac{b_5}{b_4}\right| \tag{2.43}$$

$$|w - w_2| = \frac{1}{|M|}\left|\frac{b_6}{b_4}\right| \tag{2.44}$$

where $w_1 = -L/K$ and $w_2 = -N/M$. In terms of the powers measured at the various ports, this gives

$$|w|^2 = P_3/P_4 \tag{2.45}$$

$$|w - w_1|^2 = \zeta\, P_5/P_4 \tag{2.46}$$

$$|w - w_2|^2 = \rho\, P_6/P_4 \tag{2.47}$$

In the complex w plane these are just equations of circles, as shown in Figure 2.10. So with the powers at ports 3, 4, 5 and 6 known, the problem of finding the complex quantity b_3/b_4 and reducing the six-port to an equivalent four-port is just one of finding the intersection of three circles.

The more difficult problem is that of finding the calibration constants, w_1, w_2, ζ and ρ. This requires an additional level of calibration for the six-port. First, a six-port-to-four-port calibration determines these constants, and then a four-port calibration procedure which may be similar to that described in section 2.1 completes the system calibration.

Interestingly enough, no precision calibration standards are required for the six-port-to-four-port calibration. A constraining relationship can be found between the calibration constants based solely on the linearity of the six-port network. This relationship allows the calibration constants to be determined by observing measurements of a few roughly-known reflection coefficients. The values of the calibration constants determined are independent of the particular reflection coefficient values used for the calibration.

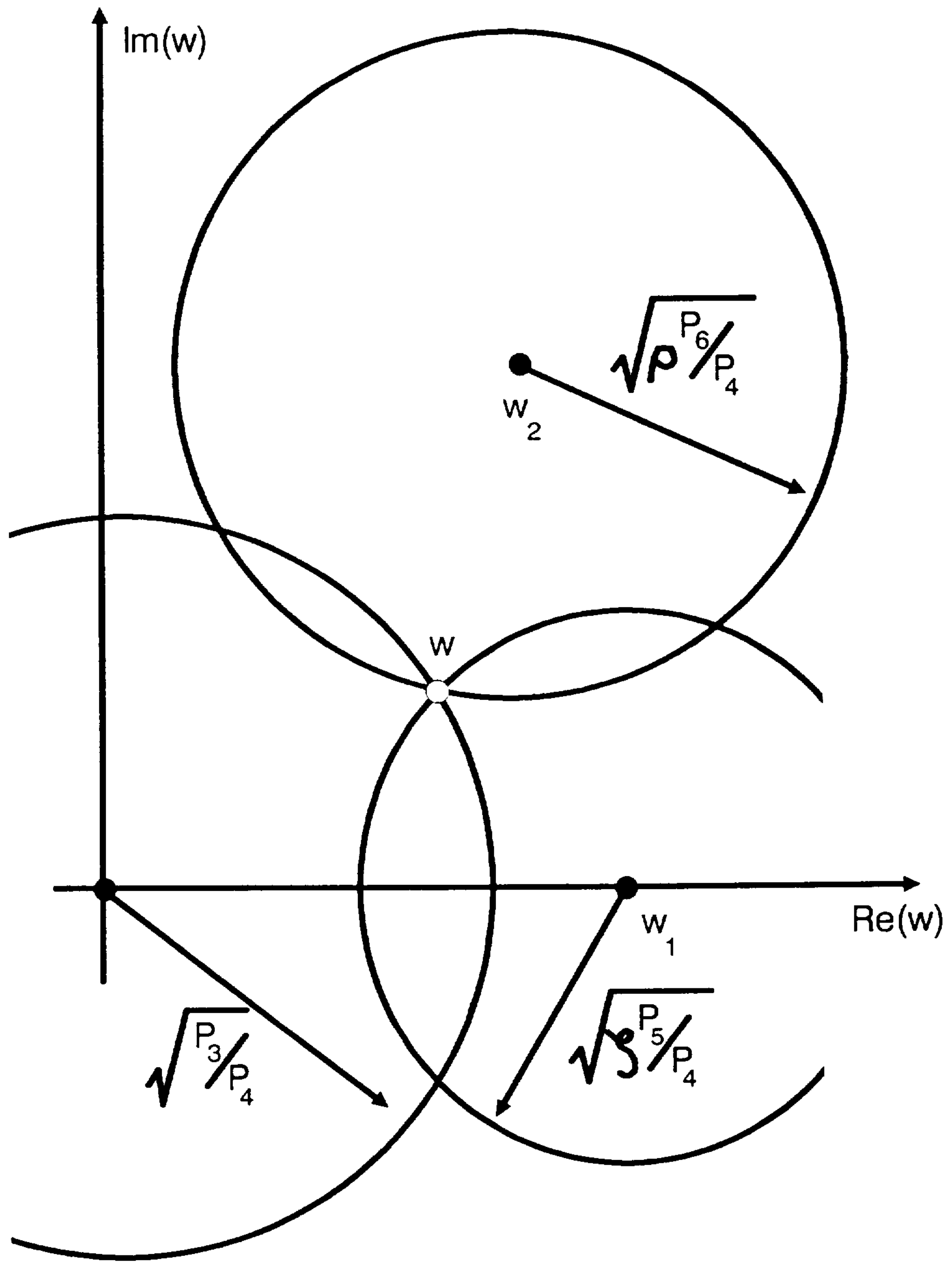


Figure 2.10 Outputs of the six-port's power detectors give a family of circles that can be solved for the complex ratio of the signals at two of the detectors.

The procedure for deriving the constraining relationship, as outlined by Engen[5], is to solve (2.46) and (2.47) to give the real and imaginary parts of w in terms of the measured powers and the calibration constants. Then, using (2.45), $(\mathrm{Re}\,w)^2 + (\mathrm{Im}\,w)^2 = P_3/P_4$ eliminates the dependence on w altogether, leaving the desired result.

Expanding (2.46) gives

$$(w - w_1)(w^* - w_1^*) = \zeta \frac{P_5}{P_4} \tag{2.48}$$

$$|w|^2 - ww_1^* - w_1 w^* + |w_1|^2 = \zeta \frac{P_5}{P_4} \tag{2.49}$$

$$|w|^2 - 2R_e\,(ww_1^*) + |w_1|^2 = \zeta \frac{P_5}{P_4} \tag{2.50}$$

Substituting (2.35) into (2.50) and expanding $R_e(ww_1{}^*)$ gives

$$\frac{P_3}{P_4} - 2\,[(\mathrm{Re}\,w)(\mathrm{Re}\,w_1) + (\mathrm{Im}\,w)(\mathrm{Im}\,w_1)] + |w_1|^2 = \zeta \frac{P_5}{P_4} \tag{2.51}$$

which is a linear equation in Re w and Im w. Performing the same operations on (2.47) gives

$$\frac{P_3}{P_4} - 2\,[(\mathrm{Re}\,w)(\mathrm{Re}\,w_2) + (\mathrm{Im}\,w)(\mathrm{Im}\,w_2)] + |w_2|^2 = \rho \frac{P_6}{P_4} \tag{2.52}$$

These two equations can be solved for Re w and Im w to give

$$\mathrm{Re}\,w = \frac{1}{2}\,\frac{(\mathrm{Im}\,w)(|w_2|^2 + P_3/P_4 - \rho P_6/P_4) + (\mathrm{Im}\,w_2)(\zeta\, P_5/P_4 - P_3/P_4 - |w_1|^2)}{(\mathrm{Re}\,w_2)\,(\mathrm{Im}\,w_1) - (\mathrm{Re}\,w_1)\,(\mathrm{Im}\,w_2)} \tag{2.53}$$

$$\mathrm{Im}\,w = -\,\frac{1}{2}\,\frac{(\mathrm{Re}\,w_1)(|w_2|^2 + P_3/P_4 - \rho P_6/P_4) + (\mathrm{Re}\,w_2)(\zeta\, P_5/P_4 - P_3/P_4 - |w_1|^2)}{(\mathrm{Re}\,w_2)\,(\mathrm{Im}\,w_1) - (\mathrm{Re}\,w_1)\,(\mathrm{Im}\,w_2)} \tag{2.54}$$

Squaring these and adding them gives, after some lengthy algebra,

$$a\left(\frac{P_3}{P_4}\right)^2 + b\zeta^2\left(\frac{P_5}{P_4}\right)^2 + c\rho^2\left(\frac{P_6}{P_4}\right)^2 + (c-a-b)\ \zeta\left(\frac{P_3P_5}{P_4^{\,2}}\right)$$

$$+ (b-a-c)\rho\left(\frac{P_3P_6}{P_4^{\,2}}\right) + (a-b-c)\ \zeta\rho\left(\frac{P_5P_6}{P_4^{\,2}}\right) \tag{2.55}$$

$$+ a(a-b-c)\frac{P_3}{P_4} + b(b-a-c)\ \zeta\frac{P_5}{P_4} + c(c-a-b)\rho\frac{P_6}{P_4} + abc = 0$$

where

$$a = |w_1 - w_2|^2 \tag{2.56}$$

$$b = |w_2|^2 \tag{2.57}$$

$$c = |w_1|^2 \tag{2.58}$$

Equation (2.55) gives the relationship between w_1, w_2, ζ and ρ and the power readings observed for any measured reflection coefficient. From here, the calibration can proceed in one of two ways. Observing nine different values of reflection coefficient, and inserting the observed values of power into (2.55) generates a 9X9 matrix, which can be solved to give the values of a/(abc), $b\zeta^2$/(abc),...,c(c-a-b) ρ/(abc). In order to find a,..., ρ, however, this third-order set of equations must be solved simultaneously, an iterative process which does not always converge to the proper values.

There is an alternative approach, pointed out by Engen [5], which yields the values of a,..., ρ in closed form with as few as five observations of reflection coefficients. The assumption that results in this simplification is that all of the values used in the calibration lie along a single circle in the w-plane. This can be achieved experimentally to a good precision through the use of a sliding short circuit. Sliding short circuits are readily available in coaxial transmission lines and rectangular waveguides. As the short circuit moves from point to point in the transmission line, it traces out a circle in the reflection coefficient plane ($|\Gamma| = 1$). A bilinear transform always carries a circle into a circle, so in the w-plane, another circle is traced out, $|w-R_c|^2 = R^2$.

The situation is shown in Figure 2.11. Only one measurement center, w_1, is considered in the first stage of the calibration. It can be assumed that this first center lies on the w-plane's real axis. This simply amounts to an arbitrary choice of the phase of b_3. Since the phase quantity to be found is the phase difference between b_3 and b_4, this does not affect the problem. Choosing the phase of w_1 sets the value of the phase of w_2, which must be found later in the procedure.

A relationship similar to (2.55) between the observed powers and the calibration constants must be found. This can be done most easily by observing that the equations that define the present case, $|w|^2 = P_3/P_4$, $|w-w_1|^2 = \zeta P_5/P_4$ and $|w-R_c|^2 = R^2$ differ from (2.45-2.47) only in the last equation. Thus, the result of eliminating w from these equations can be found by substituting R_c for w_2 and R^2 for $\rho\ P_6/P_4$ in equation (2.55). This gives

$$A\left(\frac{P_3}{P_4}\right)^2 + 2B\left(\frac{P_3\ P_5}{P_4^{\ 2}}\right) + C\left(\frac{P_5}{P_4}\right)^2 + 2D\left(\frac{P_3}{P_4}\right) + 2E\left(\frac{P_5}{P_4}\right) + F = 0 \qquad (2.59)$$

where

$$A = a' \qquad (2.60)$$

$$B = \zeta\ (c-a'-b')/2 \qquad (2.61)$$

$$C = \zeta^2 b' \qquad (2.62)$$

$$D = [R^2(b'-a'-c) + a'(a'-b'-c)]/2 \qquad (2.63)$$

$$E = \zeta[R^2(a'-b'-c) + b'(b'-a'-c)]/2 \qquad (2.64)$$

$$F = [R^4 + R^2(c-a'-b') + a'b']c \qquad (2.65)$$

and where

$$a' = |w_1 - R_c|^2 \qquad (2.66)$$

$$b' = |R_c|^2 \qquad (2.67)$$

$$c = |w_1|^2 \qquad (2.68)$$

$$\zeta = \frac{1}{|K|^2} \qquad (2.69)$$

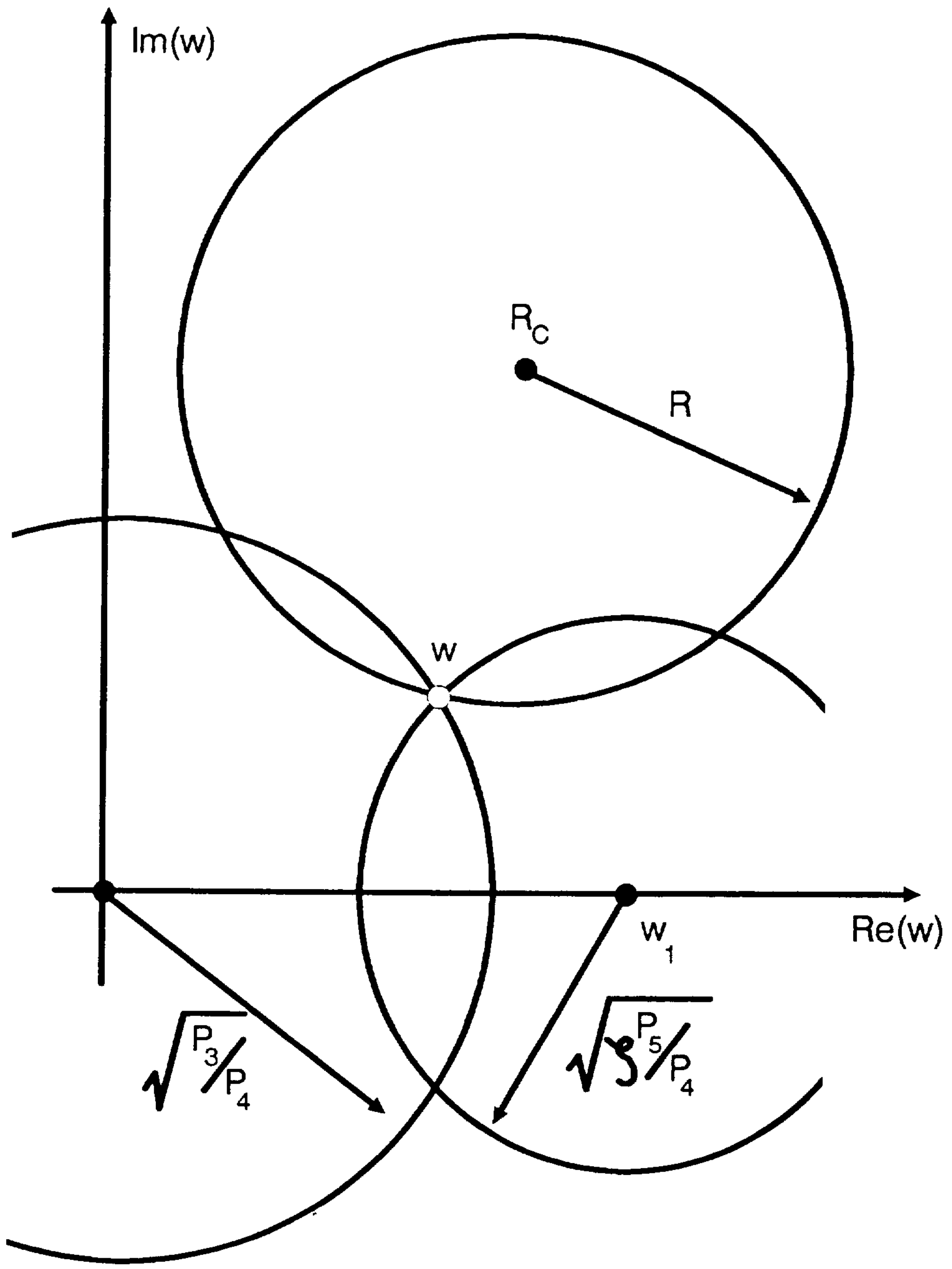


Figure 2.11 The sliding short circuit, used in the first stage of the the six-port calibration, traces out a circle in the complex w-plane.

Surprisingly, as will be seen below, (2.60--2.65) can be solved in closed form to give a′, b′, c, ζ and R^2. First, however, values of A,...,F must be determined experimentally.

Application of a standard test [6] to (2.59) shows that it is the equation of an ellipse in the P_3/P_4, P_5/P_4 plane. This ellipse is constrained to the first quadrant of the plane. Thus, A,...,F can be determined by least-squares fitting a conic section to the observed pairs (P_3/P_4, P_5/P_4), and then testing to assure that the resulting coefficients give an ellipse in the first quadrant. A six-port network with the center w_1 badly placed (e.g., w_1 very close to zero relative to the radius R) can fail these tests. In this case, the observed pairs will lie along an ellipse that is so eccentric that small measurement errors in the values of (P_3/P_4, P_5/P_4) can cause the best-fit conic section to be a hyperbola, or an ellipse crossing out of the first quadrant. A failure of this type indicates that measurement accuracy of the six-port at the frequency of the failure would be intolerably bad.

The best fit conic section can be found in a straightforward way. Substituting $x = P_3/P_4$ and $y = P_5/P_4$ into (2.59) and dividing through by F gives

$$\frac{A}{F} x^2 + 2 \frac{B}{F} xy + \frac{C}{F} y^2 + 2 \frac{D}{F} x + 2 \frac{E}{F} y + 1 = 0 \tag{2.70}$$

The error to be minimized can then be written as

$$\xi = \sum_i \left[\frac{A}{F} x_i^2 + 2 \frac{B}{F} x_i y_i + \frac{C}{F} y_i^2 + 2 \frac{D}{F} x_i + 2 \frac{E}{F} y_i + 1\right]^2 \tag{2.71}$$

To minimize this error, the partial derivatives $\partial\xi/\partial(A/F)$,..., $\partial\xi/\partial(E/F)$ are taken and set equal to zero. This yields five linear equations in the five unknowns A/F,...,E/F. Observing at least five different values of reflection coefficient, and solving a 5X5 matrix, then, yields values of A/F,...,E/F. Given these values, the task remaining is to solve (2.60 - 2.65) for a'...ζ.

This is achieved by first making the following definitions:

$$\alpha = (R^2 + a')/\zeta \tag{2.72}$$

$$\beta = [(R^2 - a')(R^2 - b') + 2R^2 c]/\zeta \tag{2.73}$$

$$\gamma = R^2 + b' \tag{2.74}$$

$$\delta = (R^2 - a')/\zeta \tag{2.75}$$

$$E = R^2 - b' \tag{2.76}$$

Substitution shows that the quantities $\alpha,\ldots,\epsilon$ can be expressed directly in terms of A,...,F:

$$\frac{BD - AE}{AC - B^2} = \alpha \tag{2.77}$$

$$\frac{DE - BF}{AC - B^2} = \beta \tag{2.78}$$

$$\frac{BE - DC}{AC - B^2} = \gamma \tag{2.79}$$

$$\frac{AF - D^2}{AC - B^2} = \delta^2 \tag{2.80}$$

$$\frac{CF - E^2}{AC - B^2} = \epsilon^2 \tag{2.81}$$

It is observed that the numerators and denominators of the left sides of (2.77 - 2.81) can be divided through by F^2 to yield expressions only in terms of A/F,...,E/F. Thus the values of $\alpha,\ldots,E^2$ can be found directly from the sliding short calibration data. The expressions for $\alpha,\ldots,E^2$, (2.72 - 2.76) can in turn be solved for a',...,R^2:

$$a' = |w_1 - R_c|^2 = \frac{(\alpha - \delta)(\gamma + \epsilon)}{2(\alpha + \delta)} \tag{2.82}$$

$$b' = |R_c|^2 = \frac{\gamma - \epsilon}{2} \tag{2.83}$$

$$c = |w_1|^2 = \frac{\beta - \delta\epsilon}{\alpha + \delta} \tag{2.84}$$

$$\zeta = \frac{\gamma + \epsilon}{\alpha + \delta} \tag{2.85}$$

$$R^2 = \frac{\gamma + \epsilon}{2} \tag{2.86}$$

So the total six-port-to-four-port reduction is a three-step process. First, the sliding short calibration data are summed into the 5X5 matrix derived from (2.71). Solving this gives A/F,...,E/F. Next, these values are used in (2.77 - 2.81) to give values of $\alpha,\ldots,\epsilon^2$. Finally, these quantities are used in (2.82 - 2.86) to give values of a',...,R^2, which contain the calibration constants wanted.

There are still problems, however. Since (2.80) and (2.81) only give expressions for δ^2 and ϵ^2, there is a sign ambiguity in the determinations of δ and ϵ. Sign ambiguity problems are inherent in the six-port network analyzer, since only magnitudes squared are measured by the instrument. Such ambiguities are encountered at several steps of calibration and measurement, and must be carefully resolved at each step to assure the validity of subsequent results.

In the present case, it is seen from the definition (2.76) that ϵ is negative if $|R_c|^2 > R^2$, that is, if the circle drawn in the w-plane does not enclose the origin. Similarly from (2.75), δ is negative if the measurement center w_1 is outside the circle drawn by the sliding short. This provides one method of resolving this sign ambiguity using knowledge of the network used. Assume first that the interior of the $|\Gamma| = 1$ circle maps to the interior of the corresponding circle in the w-plane. This is the case for most practical six-ports and can be verified by checking the values of P_3/P_4 and P_5/P_4 for some Γ, $|\Gamma|<1$ and verifying that the resulting point falls inside the sliding short's ellipse. Given this, if it is known that $|b_3|\rightarrow 0$ for some value of Γ with $|\Gamma|\leq 1$, then the $|\Gamma| = 1$ circle encloses the origin of the w-plane and $\epsilon>0$. A similar argument using b_5 can be used for δ.

This is the approach taken with the sampled-line network analyzer. The system is engineered so that none of the $|b_i|$'s can go to zero for any Γ, $|\Gamma|\leq 1$, ensuring that $\epsilon<0$ and $\delta<0$.

If the properties of the six-port network are not known, this sign ambiguity can still be resolved, through a clever technique outlined by Engen. One takes all four possible choices of sign for ϵ and δ. Using the resulting values of the calibration constants and the calibration data from the sliding short and a matched termination, four parallel calculations of the value of w_2 can be performed. All the calculations with incorrect sign assumptions yield self-contradictory results. The one with the correct δ and ϵ will yield the correct value of w_2.

Given that ϵ and δ are found correctly, there is still a sign ambiguity that can not be resolved through the equations above. This ambiguity is in the sign of Im R_c. If it is chosen incorrectly, the value found by the six-port will be w^* instead of w.

Knowledge of the properties of the six-port used provides the best way to find the sign of Im R_c. This is the approach used with sampled-line network analyzer. Otherwise, some completely independent scheme must be devised, perhaps using the reflection coefficient of an additional known standard.

With these sign ambiguities in the determination of a,...,R^2 and Im R_c resolved, there is still the problem of finding w_2 and ρ. Engen's method described above is an option here. Another option is to substitute w_2 for w_1 and ρ for ζ, in the above and redo the procedure used to find w_1.

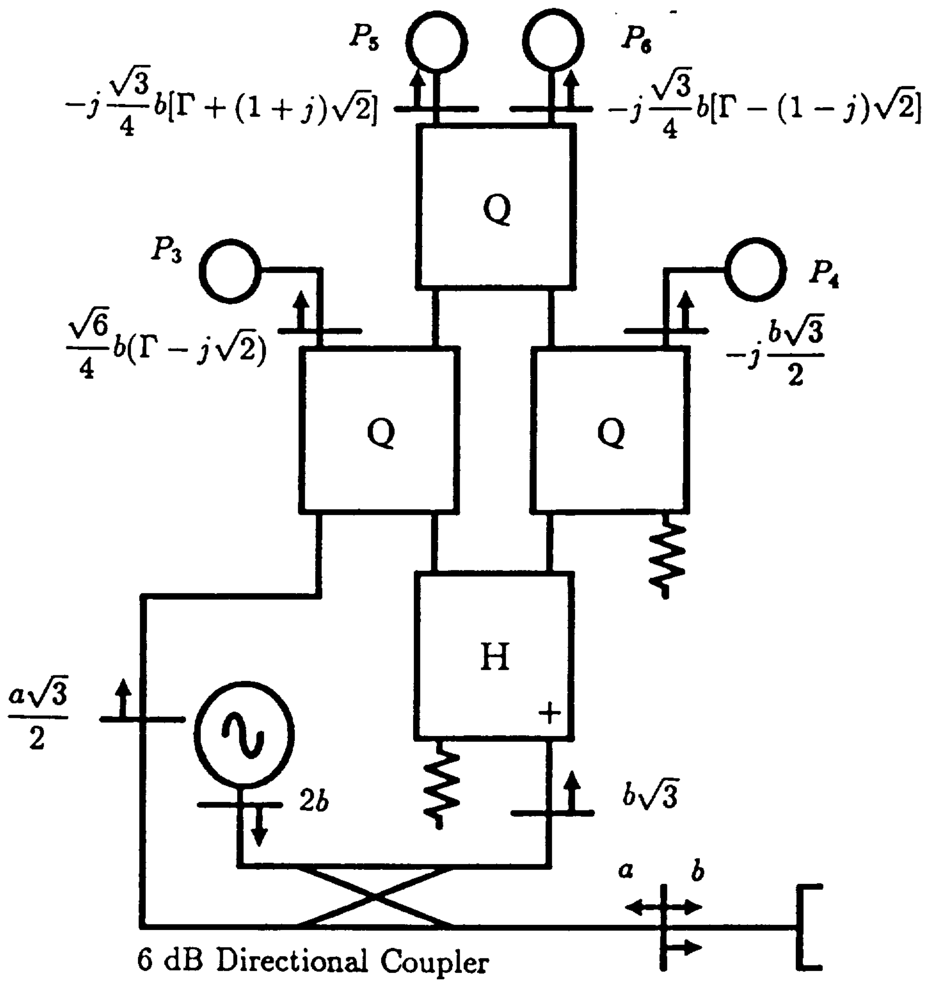
This yields w_2 on the real axis, but the value of R_c found has a different angle from that found in the determination of w_1. Rotating the second result for R_c so that it matches the first rotates w_2 to its proper angle in the plane of the first solution as well. This is the approach used in the sampled-line network analyzer. It is fairly simple and allows the results from the various detector outputs to be compared with uniform criteria in the calibration procedure.

It is interesting to note at this point that, without any knowledge of w_2 or ρ, it is possible to determine w to within a sign. As can be seen by returning to Figure 2.10, the value of w must be at one of the two intersections of the b_3/b_4 circle and the b_5/b_4 circle. The b_6/b_4 circle simply resolves this ambiguity. If, however, there is another constraint imposed by the network used, which rules out one of these intersections, then the third circle is unnecessary. Then only three detector outputs are required to determine w, and the resulting instrument is known as a five-port network analyzer. The sampled-line network analyzer has this property and is in many ways an extension of the five-port network analyzer concept.

If additional accuracy is sought in determination of the calibration constants, the values found through the closed-form approach outlined above can be used as first estimates of the coefficients of (2.45). An optimizer can then be used to minimize the sum of the squares of the values found in evaluating the left side of (2.45) for all the reflection coefficients observed in the calibration procedure.

An important determinant of system accuracy is the geometric placement of the measurement centers around the $|\Gamma| = 1$ circle in the w-plane. This placement is set by the particular six-port network used in the implementation of the network analyzer. Clearly, if all the measurement centers are close together compared to the unit circle, then triangulating from them to find Γ will yield poor accuracy. Figure 2.12 shows an implementation used at NBS [7], and the resulting placement of the measurement centers.

This implementation has the advantage that b_4 depends only on b_2, to the limit of the directivity of the hybrids used. This makes b_3/b_4 approximately a linear transform of Γ instead of a bilinear transform, which can simplify some of the preliminary calculations. The regular spacing of the measurement centers around the unit circle results in good accuracy for this instrument.



P_5
P_6
$-j\frac{\sqrt{3}}{4}b[\Gamma + (1+j)\sqrt{2}]$
$-j\frac{\sqrt{3}}{4}b[\Gamma - (1-j)\sqrt{2}]$
Q
P_3
P_4
$\frac{\sqrt{6}}{4}b(\Gamma - j\sqrt{2})$
$-j\frac{b\sqrt{3}}{2}$
Q
Q
H
+
$\frac{a\sqrt{3}}{2}$
$2b$
$b\sqrt{3}$
a
b
6 dB Directional Coupler

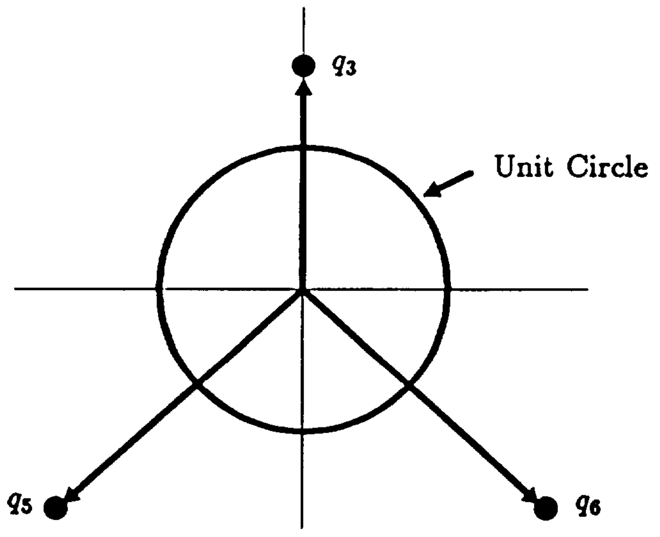


Figure 2.12 The six-port network analyzer as implemented at NBS. The boxes marked 'H' are 180° hybrid networks and those marked with 'Q' are quadrature hybrids.

As has been seen, the six-port network can be calibrated to measure the complex ratio of b_3/b_4. This ratio must then be calibrated to yield the value of $\Gamma = a_2/b_2$. As noted above, this requires determination of the three complex constants of the bilinear transform relating Γ and b_3/b_4.

If a single six-port reflectometer is used, then the same calibration options are available as with the four-port reflectometer. Three known standards are measured, and the results can be inverted to give the complex constants of the bilinear transform. If two six-ports are used, as in an S-parameter measurement system, however, other options are available, as will be seen below.

2.2.2 Dual Reflectometer Calibration - The TRL Scheme

With a single reflectometer, the final accuracy of a calibration depends on how accurately the reflection coefficients of a set of precision terminations are known. To set the reference plane, the reflection coefficient of at least one of the standards must be known to a precision greater than that of the network analyzer over the entire frequency range of operation. Even with the simplest of calibration standards, the short circuit, this is often impractical. A set of two reflectometers, however, can "calibrate each other" through a procedure developed at the National Bureau of Standards [8] known as the "thru-reflect-line" or TRL calibration procedure. In this procedure, the precise properties of the calibration standards need not be known, and are in fact derived as a by-product of the calibration. Two standards are used for this procedure. One is a precision coaxial line of approximately known length and the other is a termination with a reflection coefficient different from zero, the exact value of which is only approximately known.

As it turns out, the requirement of having two reflectometers for the TRL procedure is not a drawback. It will be seen below that two reflectometers are used to construct an S-parameter measurement system using six-ports. With two six-ports present in the system in any case, the fact that they can calibrate each other is a bonus.

The TRL calibration procedure is implemented on the sampled-line network analyzer, and will be described briefly here.

The TRL calibration procedure is only a reflectometer calibration procedure. Given two reflectometers that can measure b_3/b_4, TRL provides the coefficients of the bilinear transforms to give the Γ's connected to each reflectometer. Additional calibration is required to allow the two reflectometers to measure the S-parameters of a general two-port network.

The three measurements used for TRL are shown in Figure 2.13. As described in Section 2.1, an uncalibrated reflectometer can be represented as an ideal reflectometer with an error two-port between its port and the measurement port. This representation is used in Figure 2.13. The six-port to four-port conversion described above allows the two reflectometers to measure their respective b_3/b_4's. The goal of the TRL calibration procedure is to determine the parameters of the error boxes A and B.

In the first measurement, the two reflectometers are connected port to port. This measurement yields the cascade of the two error two-ports. In the second measurement, the two reflectometers look at each other through a length of precision transmission line, which by hypothesis contains no internal reflections or reflections from its ports. The length of this line does not need to be known exactly, but for numerical stability, a value near a quarter wavelength at the measurement frequency is desirable. In the final measurement, a nominal short circuit is connected to the two reflectometer ports.

When the two reflectometers look at each other through a given two-port, their responses in terms of that network's S-parameters can be written as follows:

$$b_1 = S_{11}a_1 + S_{12}a_2 \tag{2.87}$$

$$b_2 = S_{21}a_1 + S_{22}a_2 \tag{2.88}$$

where, for the present case $(a_1\ a_2)^T = (b_4\ b_4')^T$ and $(b_1\ b_2)^T = (b_3\ b_3')^T$. Dividing (2.87) by a_1 and (2.88) by a_2 and eliminating a_2/a_1 gives

$$w_2\ S_{11} + w_1\ S_{22} - \Delta = w_1 w_2 \tag{2.89}$$

where

$$\Delta = S_{11}\ S_{22} - S_{12}\ S_{21} \tag{2.90}$$

and

$$w_1 = b_1/a_1 = b_3/b_4 \tag{2.91}$$

$$w_2 = b_2/a_2 = b_3'/b_4' \tag{2.92}$$

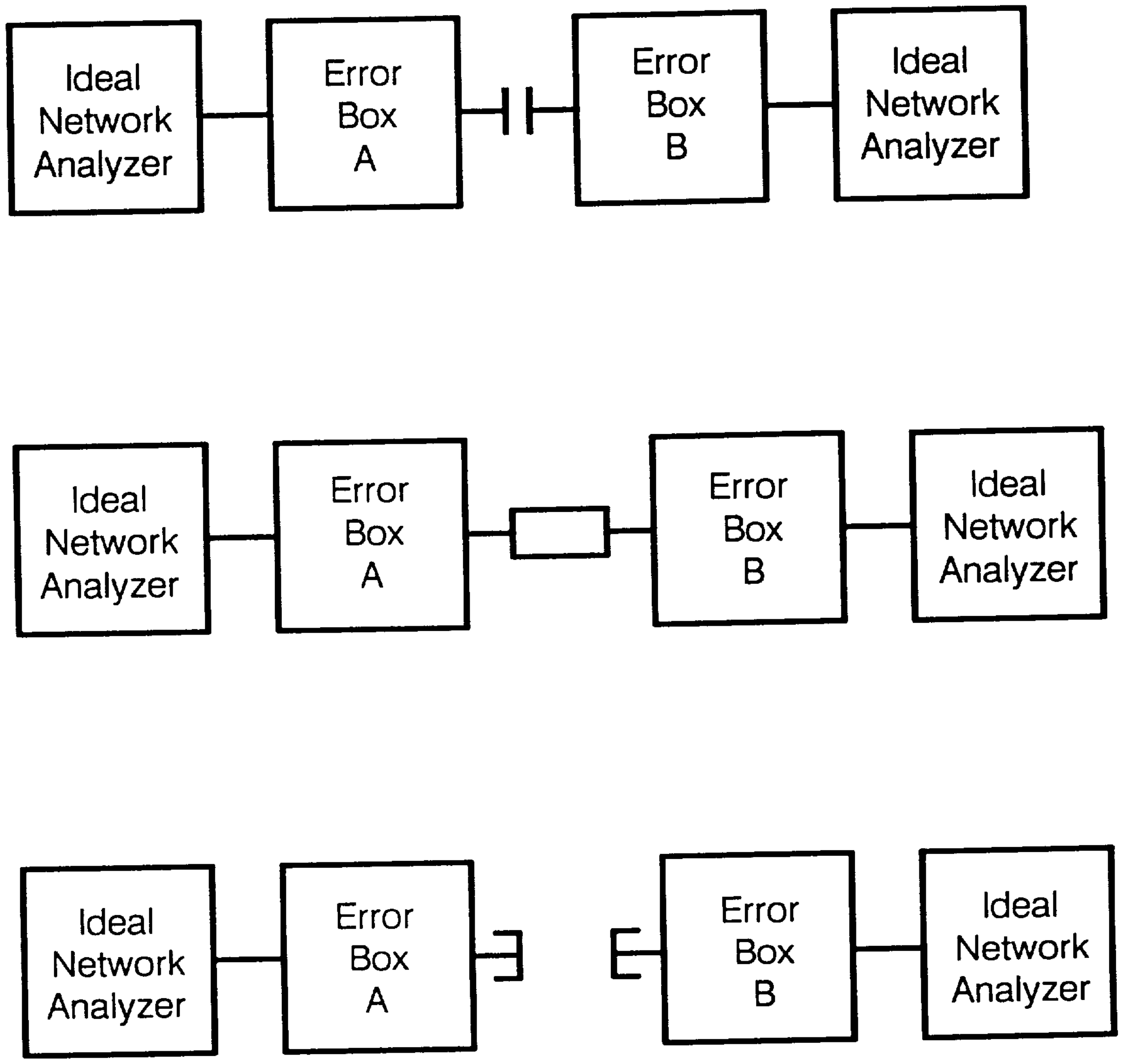


Figure 2.13 In the TRL calibration procedure, the coefficients of the error boxes A and B of two reflectometers are determined by making the three measurements shown here. A straight through, a length of line and a high-reflection load of approximately known value are used.

By observing values of w_1 and w_2 for three values of a_2/a_1, a set of three linear equations in three unknowns can be generated, and solving these gives the values of S_{11}, S_{22} and Δ. In the laboratory, the value of a_2/a_1 is changed by placing a variable phase shifter or variable attenuator between one of the reflectometer heads and the signal generator. In practice, more than three values of a_2/a_1 are used and a least-squares solution is found.

Since the TRL procedure involves cascaded two-port networks, it is convenient to work in terms of the wave cascading parameters or T-parameters. The T-parameters are defined by

$$\begin{bmatrix} b_1 \\ a_1 \end{bmatrix} = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} \begin{bmatrix} a_2 \\ b_2 \end{bmatrix} \tag{2.93}$$

$$= \overline{\overline{T}} \begin{bmatrix} a_2 \\ b_2 \end{bmatrix} \quad . \tag{2.94}$$

The T-matrix for the cascade of two networks is just the product of the T-matrices of the two networks.

The T-matrix can be written in terms of the S-matrix as

$$\begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} = \frac{1}{S_{21}} \begin{bmatrix} -\Delta & S_{11} \\ -S_{22} & 1 \end{bmatrix} \tag{2.95}$$

This result is interesting in that it shows that by solving a set of equations like (2.89) all the T-parameters of an unknown network can be found to within a multiplicative factor, $1/S_{21}$. As will be seen below, the TRL method deals only with ratios of T-parameters, so all the required information can be found in this way. With this recognized, the description of the TRL procedure may continue.

If $\overline{\overline{A}}$ and $\overline{\overline{B}}$ represent the T-matrices of the two error boxes, A and B, respectively, then for the "thru" connection, the fictional two-port $\overline{\overline{U}}$ that is observed is given by

$$\overline{\overline{U}} = \overline{\overline{A}}\,\overline{\overline{B}} \tag{2.96}$$

For the "line" connection, the observed two-port $\overline{\overline{D}}$ is

$$\overline{\overline{D}} = \overline{\overline{A}}\,\overline{\overline{L}}\,\overline{\overline{B}} \tag{2.97}$$

where $\bar{\bar{L}}$ represents the T-parameters of the line inserted between the two reflectometers. Equation (2.96) can be solved for $\bar{\bar{B}}$ to give

$$\bar{\bar{B}} = \bar{\bar{A}}^{-1} \bar{\bar{U}} \tag{2.98}$$

Inverting both sides of (2.98) and post-multiplying (2.97) by the result gives

$$\bar{\bar{D}}\, \bar{\bar{U}}^{-1}\, \bar{\bar{A}} = \bar{\bar{A}}\, \bar{\bar{L}}\, \bar{\bar{B}}\, \bar{\bar{B}}^{-1} \tag{2.99}$$

$$\bar{\bar{X}}\, \bar{\bar{A}} = \bar{\bar{A}}\, \bar{\bar{L}} \tag{2.100}$$

where

$$\bar{\bar{X}} = \bar{\bar{D}}\, \bar{\bar{U}}^{-1} \tag{2.101}$$

An assumption must be made about the value of $\bar{\bar{L}}$, the T-parameters of the length of line used as a calibration standard. It is taken to be

$$\bar{\bar{L}} \begin{pmatrix} e^{-\gamma l} & 0 \\ 0 & e^{\gamma l} \end{pmatrix} \tag{2.102}$$

This assumes that the line is uniform, not necessarily lossless, and that there are no reflections from it ($S_{11} = S_{22} = 0$).

Given this assumption, equation (2.100) can be expanded to give

$$X_{11} A_{11} + X_{12} A_{21} = A_{11} e^{-\gamma l} \tag{2.103}$$

$$X_{21} A_{11} + X_{22} A_{21} = A_{21} e^{-\gamma l} \tag{2.104}$$

$$X_{11} A_{12} + X_{12} A_{22} = A_{21} e^{\gamma l} \tag{2.105}$$

$$X_{21} A_{12} + X_{22} A_{22} = A_{22} e^{\gamma l} \tag{2.106}$$

Dividing (2.103) by (2.104) and (2.105) by (2.106) gives

$$\frac{X_{21}}{X_{12}}\left[\frac{A_{11}}{A_{21}}\right]^2 + \frac{X_{22} - X_{11}}{X_{12}}\left[\frac{A_{11}}{A_{21}}\right] - 1 = 0 \tag{2.107}$$

$$\frac{X_{21}}{X_{12}}\left[\frac{A_{12}}{A_{22}}\right]^2 + \frac{X_{22} - X_{11}}{X_{12}}\left[\frac{A_{12}}{A_{22}}\right] - 1 = 0 \tag{2.108}$$

So it is seen that the values A_{11}/A_{21} and A_{12}/A_{22} are solutions of the same quadratic equation. The coefficients of this equation can be determined from the measured b_3/b_4's for the thru and line measurements via (2.89) and (2.101).

It is easy to see that $(A_{11}/A_{21}) \neq (A_{12}/A_{22})$. From (2.95) this would require that the error box A have $S_{12}S_{22} = 0$ so there would be no transmission through the box.

Thus (A_{11}/A_{21}) and (A_{12}/A_{22}) are the two distinct roots of the quadratic equation. The question of root choice then arises again: which root represents which ratio? Approximate knowledge of the properties of the line calibration standard can be used to answer this question.

Dividing (2.106) by (2.104) gives

$$e^{2\gamma l}\ \frac{X_{21}\ \left(A_{12}/A_{22}\right) + X_{22}}{X_{12}\ \left(A_{21}/A_{11}\right) + X_{11}} \tag{2.109}$$

so if the length of the transmission line used as the line standard is known fairly accurately, the value of $e^{2\gamma l}$ can be calculated, and this can be used to resolve the root ambiguity. This method has the advantage that it depends only on quantities found in the course of the TRL calibration procedure, but unfortunately it is not foolproof. As stated above, the transmission line standard is chosen to have a phase delay near $\pm 90^\circ$. It is a low-loss line, so $|e^{2\gamma l}| \simeq 1$. This places $e^{2\gamma l}$ near ± 1. The two possible root choices, when inserted in the right side of (2.109), yield values that are reciprocals of each other. If the phase delay of the line is too near $\pm 90^\circ$, then $e^{2\gamma l}$ and its reciprocal cannot be easily distinguished, and measurement errors may lead to an improper root choice.

An alternative approach uses the result found in Section 2.1, that for the reflectometer with error box A,

$$w_1 = \frac{a\Gamma + b}{c\Gamma + 1} \tag{2.110}$$

where Γ is the reflection coefficient measured by the reflectometer and

$$a = A_{11}/A_{22} \tag{2.111}$$

$$b = A_{12}/A_{22} \tag{2.112}$$

$$c = A_{21}/A_{22} \tag{2.113}$$

A rough calibration using three impedance standards, such as a short circuit, an open circuit and a matched load, can be performed to find approximate values of a, b, and c. Only one of the roots of (2.107) and (2.108) will be close to b, and this serves to resolve the root ambiguity. This method does not suffer the problems of the previous one, but does require additional known, independent calibration standards, which are not easy to come by in all types of transmission line.

Inspecting (2.110 - 2.113), it is seen that b and a/c have been determined from the roots of the quadratic equation (2.103). All that remains to characterize error box A is to determine the value of a. Rearranging (2.106) gives

$$a = \frac{w_1 - b}{\Gamma(1 - w_1 c/a)} \tag{2.114}$$

So, if one uses one precision standard, say a short circuit, then (2.114) can be solved for a and the calibration of error box A is complete. This approach is known as the "thru-short-delay," or TSD calibration procedure. Error box B can then be determined from (2.96).

For the TRL procedure it is assumed that Γ is not known precisely. More manipulation shows that this knowledge is not required for the determination of a. Equation (2.96) can be rewritten as

$$A_{22}\ B_{22} \begin{pmatrix} a & b \\ c & 1 \end{pmatrix} \begin{pmatrix} \alpha & \beta \\ \gamma & 1 \end{pmatrix} = U_{22} \begin{pmatrix} d & e \\ f & 1 \end{pmatrix} \tag{2.115}$$

where

$$a = A_{11}/A_{22} \tag{2.116}$$

$$b = A_{12}/A_{22} \tag{2.117}$$

$$c = A_{21}/A_{22} \tag{2.118}$$

$$\alpha = B_{11}/B_{22} \tag{2.119}$$

$$\beta = B_{12}/B_{22} \tag{2.120}$$

$$\gamma = B_{21}/B_{22} \tag{2.121}$$

$$d = U_{11}/U_{22} \tag{2.122}$$

$$e = U_{12}/U_{22} \tag{2.123}$$

$$f = U_{21}/U_{22} \tag{2.124}$$

By premultiplying both sides of (2.115) by $\overset{=}{A}^{-1}$ and expanding, it can be shown that

$$\gamma = \frac{f - dc/a}{1 - ec/a} \tag{2.125}$$

$$\beta/\alpha = \frac{e - b}{d - bf} \tag{2.126}$$

$$a\alpha = \frac{d - bf}{1 - ec/a} \tag{2.127}$$

All the quantities on the right side of (2.125 - 2.127) are known, so γ, β/α and $a\alpha$ can be determined. A relationship like (2.114) holds for error box B. It is

$$\alpha = \frac{w_2 + \gamma}{\Gamma(1 + w_2\beta/\alpha)} \tag{2.128}$$

Assuming that the same Γ is connected to both reflectometers during the calibration procedure, Γ can be eliminated from (2.114) and (2.128) to give

$$a/\alpha = \frac{(w_1 - b)\ (1 + w_2\beta/\alpha)}{(w_2 + \gamma)\ (1 - w_1 c/a)} \tag{2.129}$$

and finally, (2.129) can be combined with (2.127):

$$a = \pm\sqrt{\frac{(w_1 - b)\ (1 + w_2\beta/\alpha)\ (d - bf)}{(w_2 + \gamma)\ (1 - w_1 c/a)\ (1 - ec/a)}} \tag{2.130}$$

and

$$\alpha = \frac{d - bf}{a(1 - ec/a)} \tag{2.131}$$

The sign ambiguity in (2.129) can be resolved by approximate knowledge of Γ. Use of a short circuit for Γ is convenient here. Good short circuit standards are readily available in coaxial lines and waveguides, and can be constructed without great difficulty in many of the transmission lines used in microwave ICs.

So the final result of the above is that two reflectometers can calibrate each other without the use of high-precision calibration standards. The one big assumption made in the preceding development is that the transmission line standard used is reflectionless. The effects of reflections in this standard have not been fully evaluated, but experimental results at the National Bureau of Standards indicate that the TRL procedure yields a very accurate calibration. Typical errors estimated at $\pm$0.001 dB in measuring a 20 dB attenuator have been reported [9].

2.2.3 S-Parameter Measurements with the Six-Port Network Analyzer

Figure 2.14 shows how two six-port network analyzers are used to construct an S-parameter measurement system. It is assumed here that both six-ports have been calibrated to read the true ratios, Γ_1 ($=b_1/a_1$) and Γ_2 ($=b_2/a_2$) at the reference planes 1 and 2. This calibration may proceed through any of the methods outlined in the previous sections.

Through a procedure identical to that given in equations (3.50 - 3.55) above, then, the values of S_{11}, S_{22} and Δ for the unknown two-port can be determined by observing the values of Γ_1 and Γ_2 for at least three different excitations (positions of the phase shifter or attenuators).

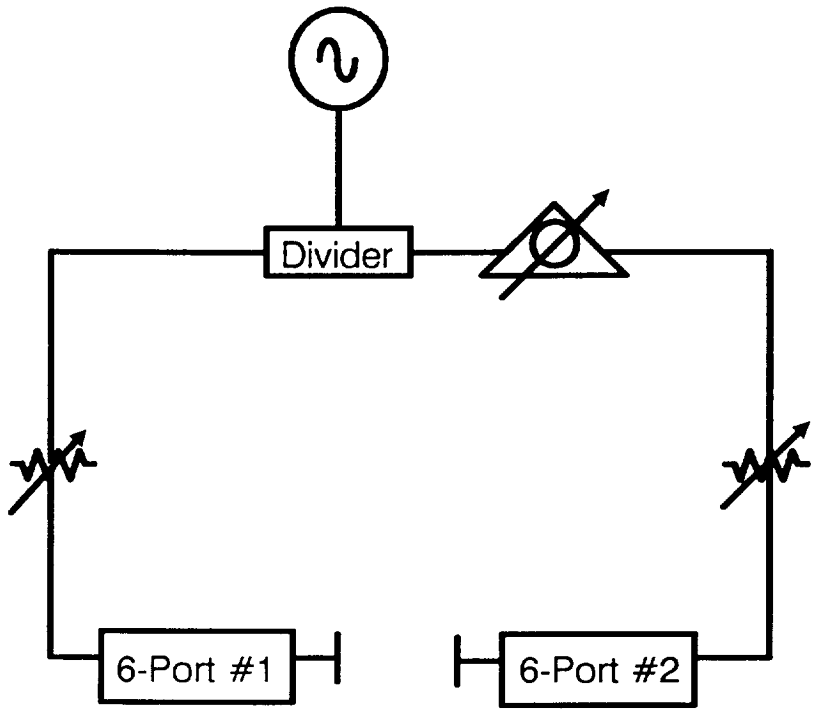


Figure 2.14 A full S-parameter measurement system using six-port network analyzers.

In order to find the complete S-parameters, some way of separating Δ to give S_{12} and S_{21} must be found. Rewriting (2.87 - 2.88) gives

$$S_{12} = (\Gamma_1 - S_{11}) \frac{a_1}{a_2} \tag{2.132}$$

$$S_{21} = (\Gamma_2 - S_{22}) \frac{a_2}{a_1} \tag{2.133}$$

so if the value of a_2/a_1 can be found for each measurement, then S_{12} and S_{21} can be found. Some additional calibration is necessary for this determination.

Consider the measurement system of Figure 2.14 as a three-port network with its port 3 near the generator and ports 1 and 2 being the measurement reference planes as numbered. Then it can be shown [7] that

$$\frac{a_2}{a_1} = \left(s_{21} - s_{11} \frac{s_{23}}{s_{13}} \right) \Gamma_1 - \left(s_{12} \frac{s_{23}}{s_{13}} - s_{22} \right) \Gamma_2 \frac{a_2}{a_1} + \frac{s_{23}}{s_{13}} \qquad (2.134)$$

where the s_{ij}'s are the S-parameters of the measurement system three-port. This can be rewritten as

$$\frac{a_2}{a_1} = C_1 \Gamma_1 - C_2 \Gamma_2 \frac{a_2}{a_1} + C_3 \qquad (2.135)$$

which can be rearranged to give

$$\frac{a_2}{a_1} = \frac{C_3 + C_1 \Gamma_1}{1 + C_2 \Gamma_2} \qquad (2.136)$$

With equation (2.136), then, a_2/a_1 can be determined from the observed Γ's for any measurement. The remaining difficulty is to find the values of the C's. This can be done by noting that equation (2.135) is linear in the C's. Using at least three known values of Γ_1, Γ_2 and a_2/a_1, a system of linear equations is formed which can be solved for the C's.

The values of the Γ's to plug into this set of equations are, of course, directly available from the measurements. The values of a_2/a_1 must be derived somewhat indirectly. This is done by measuring a set of calibration standards for which the S-parameters are approximately known. A set of precision transmission lines of approximately known length is a common choice. These lines are reciprocal, and so their complete S-parameters can be found from the knowledge of S_{11}, S_{22} and Δ which is found as noted above.

$$|S_{12}| = |S_{21}| = \sqrt{|\Delta - S_{11} S_{22}|}$$

$$\arg(S_{12}) = \arg(S_{21}) = \frac{\arg(\Delta - S_{11} S_{22})}{2} + n\pi$$

The n π in equation (2.138) results from the sign ambiguity of the square root. Since in this case, the length of each of the calibration standards is approximately known, this sign ambiguity can be resolved by calculating the expected value of arg (S_{12}) and choosing the sign that gives the value closest to this.

With all the S-parameters for the calibration standards thus measured, equations (2.132) or (2.133) can be used to find a_2/a_1 for each measurement, and determination of the values of the C's can proceed.

The values of the C's change when the phase shifter or variable attenuators are switched. Thus C's must be calculated and stored for all possible configurations of the measurement system.

After all calibration has been completed, measurement of the S-parameters of an unknown two-port proceeds as follows: the measurement system is stepped through all combinations of the phase shifter and attenuator positions and the values of the Γ's resulting are stored. These are then summed into a matrix and least-squares estimates of S_{11}, S_{22} and Δ for the unknown network are determined through a set of equations like (2.89). Then the C's are used to find a_2/a_1 for each of the measurement configurations, and calculate S_{12} and S_{21} by solving equations (2.132) and (2.133), respectively. The resulting values of S_{12} and S_{21}, respectively, are averaged to yield the final estimates of these quantities.

This is the general procedure. If it is known that the two-port being measured is reciprocal, then greater accuracy can be achieved by using (2.137-2.138) to find $S_{12} \equiv S_{21}$. Then the results from (2.132-2.133) and (2.136) can be used only to resolve the sign ambiguity in (2.138).

By changing the positions of the variable attenuators, the value of a_2/a_1 can be made arbitrarily large or small. This can be used to advantage in some measurement situations. In measuring an amplifier, for example, the signal on the output side of the amplifier is at a much higher level than that on the input side. Most six-port reflectometers have their best accuracy when measuring values of Γ for which $|\Gamma| \leq 1$. By making $|a_2/a_1|$ approximately equal to the gain of the amplifier, ratios near unity will be measured by both the input and output reflectometers.

In summary, two six-port reflectometers can be combined in a system that can measure the full S-parameters of unknown two-ports. This system has none of the switch-dependency problems that can occur with the four-port-based systems. As long as the various switches (attenuators and phase shifters) in the six-port system are repeatable, their effects are calibrated out.

2.3 Sampled-Line Network Analyzer Theory

The sampled-line network analyzer is a particularly simple implementation of the six-port type of network analyzer. It removes the need for the broad band directional couplers used in the NBS implementations. Also, the sampled-line analyzer uses more detector diodes than previous implementations and the additional data from these detectors can be used to advantage.

Figure 2.15 shows the sampled-line network analyzer schematically. It consists of a uniform transmission line with several power detector diodes connected in shunt across it. The diodes are resistively isolated from the line to minimize loading effects. An attenuator placed between the line and the device under test prevents deep voltage nulls from occurring on the line. Removal of such nulls is important for the instrument's accuracy, as will be shown below.

The diodes sample the magnitude of the microwave voltage at their points of connection. The incident and reflected waves, travelling in opposite directions on the line, are sampled with different relative phases at different points on the line. A six-port type analysis can be applied to the resulting diode voltages: one diode voltage is chosen as the denominator of the complex ratio w, and all the others are divided by that voltage. The resulting ratios give the radii of circles as in Figure 2.10. For n diodes, however, there are (n-1) circles, all ideally intersecting at a single point in the w-plane. As will be seen below, two circles are used for the determination of the reflection coefficient with the sampled-line network analyzer. The two circles used are those from the three diodes with optimum spacing for the frequency range under consideration. The ambiguity from the two intersections of the circles is resolved by knowledge of the device under test or by use of an external attenuator.

The diode spacing scheme shown in Figure 2.15 allows extension of the network analyzer's operation over a broad frequency range. As described above, a set of three detectors provides two circles that can be used to determine w to within a sign. One circle has its center at zero in the w-plane, and the other has center w_1. Assuming that some way can be found to resolve the sign ambiguity, then the requirement for an accurate measurement is that w_1 be placed in such a way that for any measured Γ, the resulting circles, centered on zero and w_1, do not intersect at too oblique an angle.

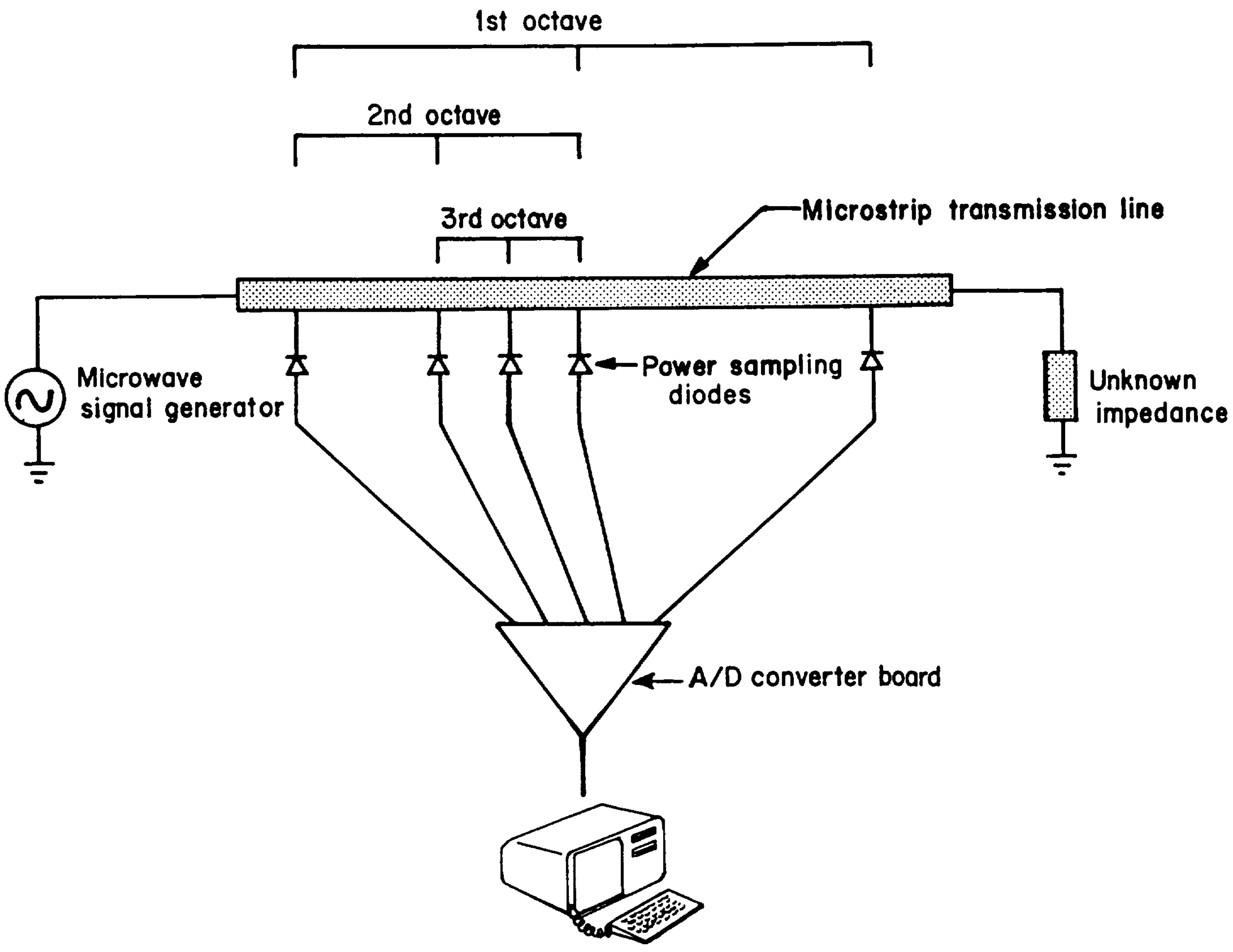


Figure 2.15 The sampled-line analyzer consists of a number of power detectors distributed along a transmission line. The spacing scheme shown allows extension of the operation of the analyzer over a wide frequency range.

Placement of w_1 in the sampled-line analyzer is determined by the spacing of the sampling diodes. A convenient measure of the "goodness" of a particular configuration is the sensitivity of the calculated value of Γ for a given measurement to noise on the detector voltages. The average noise sensitivity in the determination of Γ, over all $|\Gamma| \leq 1$ was calculated for a wide range of diode spacings. The results showed that for a given frequency, the minimum average noise sensitivity is achieved when the three diodes are uniformly spaced along the line with one sixth of a wavelength between each pair of diodes.

It turns out that one such diode triple provides good accuracy (noise sensitivity within a factor of two of the minimum) over about an octave of frequency. When one triple's useful frequency is exceeded, another triple can be formed by placing a fourth diode halfway between one pair of diodes in the original triple. This second triple has a frequency range twice as high as the first, and this method can be continued, to extend the analyzer's operating range to the upper working frequency of the diode detectors.

This is the fundamental difference between the sampled-line analyzer and previous six-port analyzer configurations. Previous analyzers used a fixed number of diodes. To extend their operating bandwidth a broader-band coupling structure was required to provide signals to the diodes. In the sampled-line analyzer, a simple resistive coupling structure is used. To extend the analyzer's frequency of operation, additional diodes are added to the system.

Sampled lines have, of course, been used for impedance measurements for many years. These instruments used sums and differences of powers measured at points along the line to determine the reflection coefficient of the device under test. A six-port type analysis could not be applied to the sum and difference data and the loading effects of the samplers were not calibrated out. Also, since ratios of the measured powers were not used, the measurements were affected by fluctuations in the signal source's output power.

2.3.1 Placement of the Measurement Centers

Given that the sampled-line analyzer is a type of six-port, the best way to examine its operation is to consider the placement in the w-plane of the measurement centers w_1, w_2,..., $w_{(n-2)}$ where n is the number of detectors used.

Figure 2.16 shows the theoretical model of the sampled-line analyzer used to calculate the positions of the measurement centers and scale factors. It is assumed that the detectors do not load the transmission line at all. The θ's are the electrical lengths of the various sections of transmission line, and the detectors are numbered as shown. Only four detectors are shown here, though there may in general be many more. Detectors 3, 4, and 5 are assumed to have the uniform $\lambda/6$-spacing, and detector i has some arbitrary placement on the line. No attenuator is placed between the line and the device under test for this analysis.

As in section 2.2,

$$b_5 = K\, b_3 + L b_4 \qquad (2.139)$$

$$\frac{1}{|K|^2}\left|\frac{b_5}{b_4}\right|^2 = \left|\frac{b_3}{b_4} + \frac{L}{K}\right|^2 \qquad (2.140)$$

$$\zeta_1 \frac{P_5}{P_4} = |w - w_1|^2 \qquad (2.141)$$

The values of $\zeta_1 = 1/|K|^2$ and $w_1 = -L/K$ are to be found.

The voltage measured by a given sampler is the sum of the left and right travelling voltage waves on the line at the point where the sampler is connected. Associating this complex voltage with a single travelling wave value, b_i, may seem odd, but it is easy to see that such an assignment is valid. The sampler could be replaced with an ideal voltage amplifier which would sample the voltage on the line and produce a travelling wave at its output with an amplitude equal to the sampled voltage. Keeping this in mind, then, expressions for $b_3,\ldots,b_6$ can be written:

$$b_4 = b_2(1 + \Gamma e^{-j2\theta_4}) \qquad (2.142)$$

$$b_3 = b_2(1 + \Gamma e^{-j2\theta_3})\, e^{-j(\theta_4 - \theta_3)} \qquad (2.143)$$

$$b_5 = b_2(1 + \Gamma e^{-j2\theta_5})\, e^{-j(\theta_4 - \theta_5)} \qquad (2.144)$$

$$b_6 = b_2(1 + \Gamma e^{-j2\theta_6})\, e^{-j(\theta_4 - \theta_6)} \qquad (2.145)$$

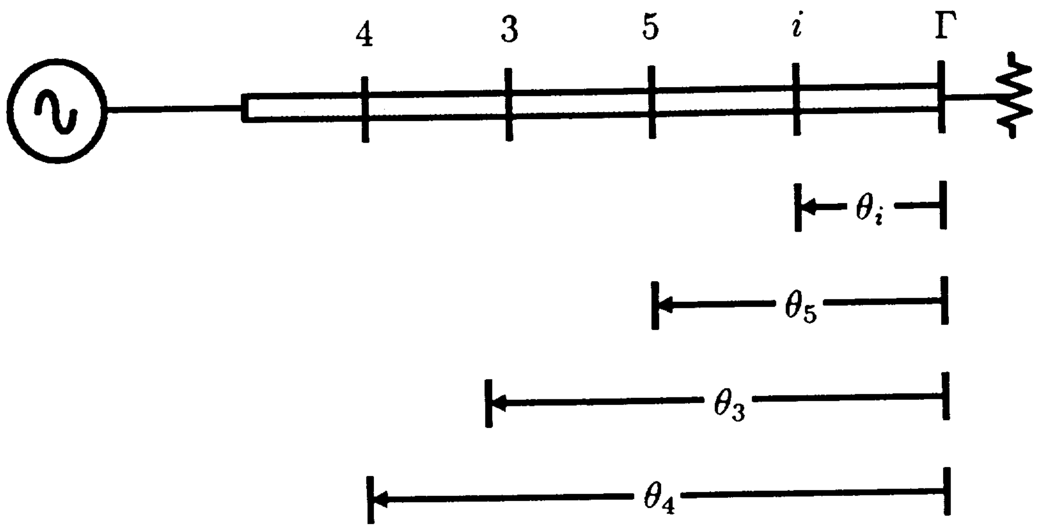


Figure 2.16 Model of the ideal sampled-line at a single frequency. The detectors have infinite impedance. Their response is the square of the magnitude of the voltage at the point of connection on the line.

Substituting (2.142-2.144) in (2.139) gives

$$(1+\Gamma e^{-j2\theta_5})\, e^{-j(\theta_4-\theta_5)} = K(1+\Gamma e^{-j2\theta_3})\, e^{-j(\theta_4-\theta_3)} + L(1 + \Gamma e^{-j2\theta_4}) \qquad (2.146)$$

$$e^{-j(\theta_4-\theta_5)} + \Gamma e^{-j(\theta_4+\theta_5)} = Ke^{-j(\theta_4-\theta_3)} + K\Gamma e^{-j(\theta_4+\theta_3)} + L+L\Gamma\, e^{-j2\theta_4} \qquad (2.147)$$

Since (2.147) must be true for all values of Γ, it can be rewritten as two equations:

$$e^{-j(\theta_4 - \theta_5)} = Ke^{-j(\theta_4 - \theta_3)} + L \qquad (2.148)$$

$$e^{-j(\theta_4 + \theta_5)} = Ke^{-j(\theta_4 + \theta_3)} + L\, e^{-j2\theta_4} \qquad (2.149)$$

When these two equations are solved for K and L they give

$$K = \frac{\sin\,(\theta_4 - \theta_5)}{\sin\,(\theta_4 - \theta_3)} \qquad (2.150)$$

$$L = \frac{\sin\,(\theta_3 - \theta_5)}{\sin\,(\theta_3 - \theta_4)} \qquad (2.151)$$

and thus

$$w_1 = -\frac{L}{K} = \frac{\sin\,(\theta_3 - \theta_5)}{\sin\,(\theta_4 - \theta_5)} \qquad (2.152)$$

$$\zeta_1 = -\frac{1}{|K|^2} = \frac{\sin^2(\theta_4 - \theta_3)}{\sin^2(\theta_4 - \theta_5)} \qquad (2.153)$$

An identical analysis for detector i, placed an electrical distance of θ_i from the load gives

$$w_{(i - 4)} = \frac{\sin\,(\theta_3 - \theta_i)}{\sin\,(\theta_4 - \theta_i)} \qquad (2.154)$$

$$\zeta_{(i - 4)} = \frac{\sin^2(\theta_4 - \theta_3)}{\sin^2(\theta_4 - \theta_i)} \qquad (2.155)$$

Finally, from (2.142-2.144), the expression for $w = b_3/b_4$ can be written:

$$\frac{b_3}{b_4} = \frac{1 + \Gamma e^{-j2\theta_3}}{1 + \Gamma e^{-j2\theta_4}} e^{-j(\theta_4 - \theta_3)} \qquad (2.156)$$

Figure 2.17 shows these results graphically. The circles of constant $|\Gamma|$ show how the bilinear transform maps the Γ-plane to the w-plane. First, $\Gamma = 0$ is mapped to $w = e^{-j(\theta_4 - \theta_3)}$. Since detectors 3 and 4 are assumed to be $\lambda/6$, the zero reflection coefficient point maps to $e^{-j\pi/3}$.

The $\Gamma = 0$ point maps to a unity-magnitude w for any diode spacing. The spacing determines the angle of the resulting w. This must be the case since, when $\Gamma = 0$, there is no standing wave on the sampled-line, and all detectors observe signals of the same magnitude. The ratio of samples at any two points on the line then has unity magnitude.

The $|\Gamma| = 1$ circle is mapped to the real axis of the w-plane. This can be seen by substituting $\Gamma = e^{j\phi}$ into (2.156). The result is

$$W = \frac{\cos(\theta_3 - \phi/2)}{\cos(\theta_4 - \phi/2)} \qquad (2.157)$$

This equation shows that $w \rightarrow 0$ when there is a standing wave null at sampler 3, and $w \rightarrow \infty$ when there is a null at sampler 4, as must be the case for the sampled-line.

Completing the picture, it is observed from equations (2.152) and (2.154) that all the measurement centers, $w_1,\ldots,w_{(n-2)}$, lie along the real axis as well. This can also be seen by examination of the physical configuration. From equation (2.141) it is seen that $w = w_1$ is the point at which P_5 goes to zero. On the sampled-line, a zero detector output indicates a standing wave null. Only unity-magnitude Γ's produce nulls, and the corresponding w lies on the real w-axis, as noted above.

So the ideal sampled-line network analyzer transforms the Γ plane such that the region of $|\Gamma|>1$ is mapped into the upper half of the w-plane (Im w > 0), and $|\Gamma|<1$ is mapped to the Im w<0 region. This has the advantage that when measuring passive circuits, only the data from three detectors need be used to find the value of Γ. These data give two circles, which will intersect at two points. The ambiguity in the value of w is resolved immediately, however, since only one of these intersections, the one with Im w $\leq$ 0, corresponds to a realizable Γ. As noted in section 2.2, this makes the sampled-line network analyzer a "five-port" network analyzer.

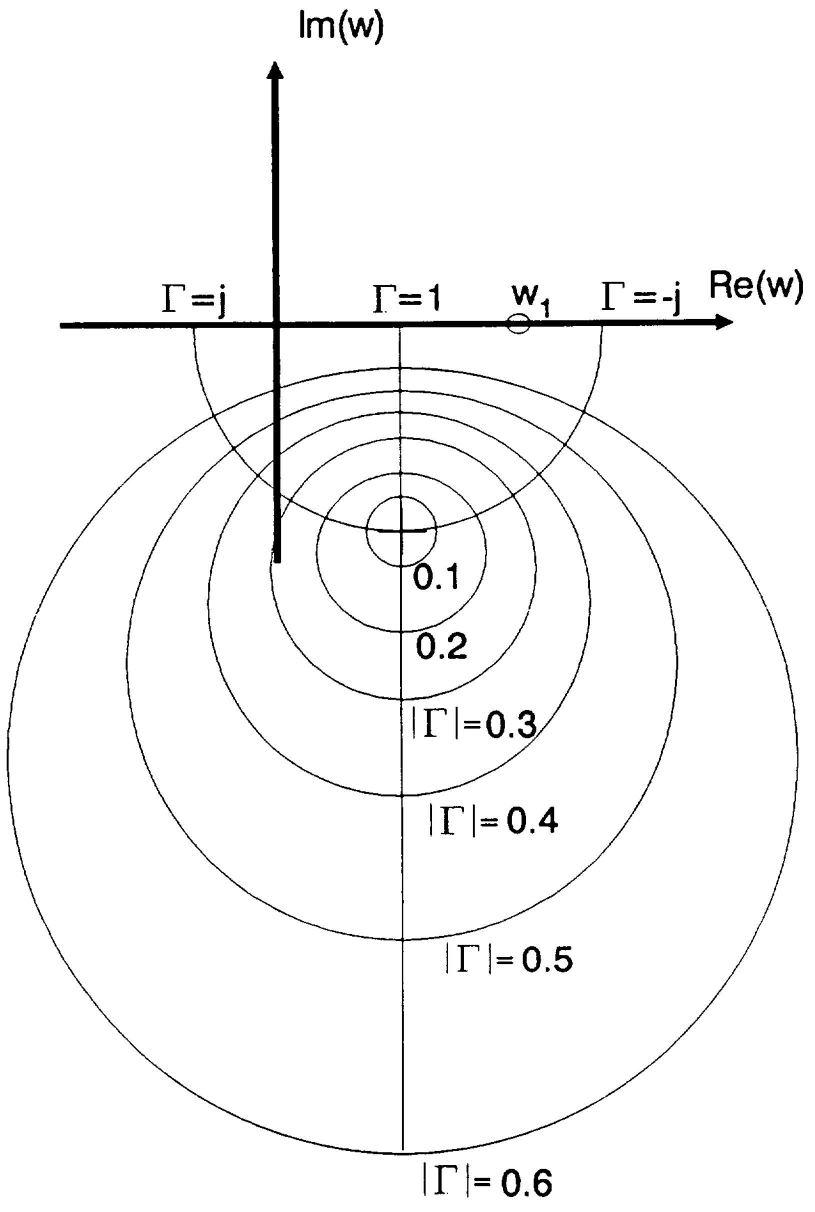


Figure 2.17 The sampled-line analyzer maps the Γ-plane to the w-plane as shown here.

Having the measurement centers lie along a line in the w-plane is a disadvantage. If it is not known whether the measured Γ has a magnitude greater than or less than one, then the ambiguity cannot be resolved from the diode data directly. Losses in the line and reflections from the diodes do cause the centers not to lie exactly along a line, but the scatter is usually small and cannot be used reliably to resolve the ambiguity. One method which could be used to resolve the ambiguity would be to place a directional coupler between the signal generator and the sampled-line and read the forward- (or reverse-) going wave with a detector. The directional detector's output could then be entered into the calibration procedure, yielding a measurement center off the real w-axis.

There is another problem, with the sampled-line analyzer as shown in Figure 2.16. Its measurement accuracy for $|\Gamma| \simeq 1$ is bad. Re-examining Figure 2.17, for a value of w near the real axis, all the circles resulting from the measured data intersect at very oblique angles. Thus, a small error due to circuit noise in one of the circles' radii results in a large error in the determination of the point of intersection.

This problem is reduced by placing an attenuator between the device under test and the sampled-line as noted above. The results are shown in Figures 2.18 and 2.19. These show contour plots of error sensitivity in the Γ-plane. For three $\lambda/6$-spaced diodes, define Re $\Gamma = f(v_1,v_2,v_3)$ and Im $\Gamma = g(v_1,v_2,v_3)$ where v_1, v_2, and v_3 are the observed detected voltages. Then the quantity plotted in the following figures is

$$\begin{aligned} E = {} & \left(\frac{\partial f}{\partial v_1}\right)^2 + \left(\frac{\partial f}{\partial v_2}\right)^2 + \left(\frac{\partial f}{\partial v_3}\right)^2 \\ & + \left(\frac{\partial g}{\partial v_1}\right)^2 + \left(\frac{\partial g}{\partial v_2}\right)^2 + \left(\frac{\partial g}{\partial v_3}\right)^2 \end{aligned} \tag{2.158}$$

These plots are in the Γ-plane. The value of w, with noise added, is found and then transformed into the Γ-plane. It is interesting to note how the transformation process symmetrizes the error around the origin of the Γ-plane. The effect of the attenuator is interesting, as well. The error function is bowl-shaped, rising steeply at the edges. Adding attenuation between the analyzer and the device under test degrades measurement accuracy slightly at the minimum of the bowl, but it flattens out the bowl, improving the accuracy near the edge.

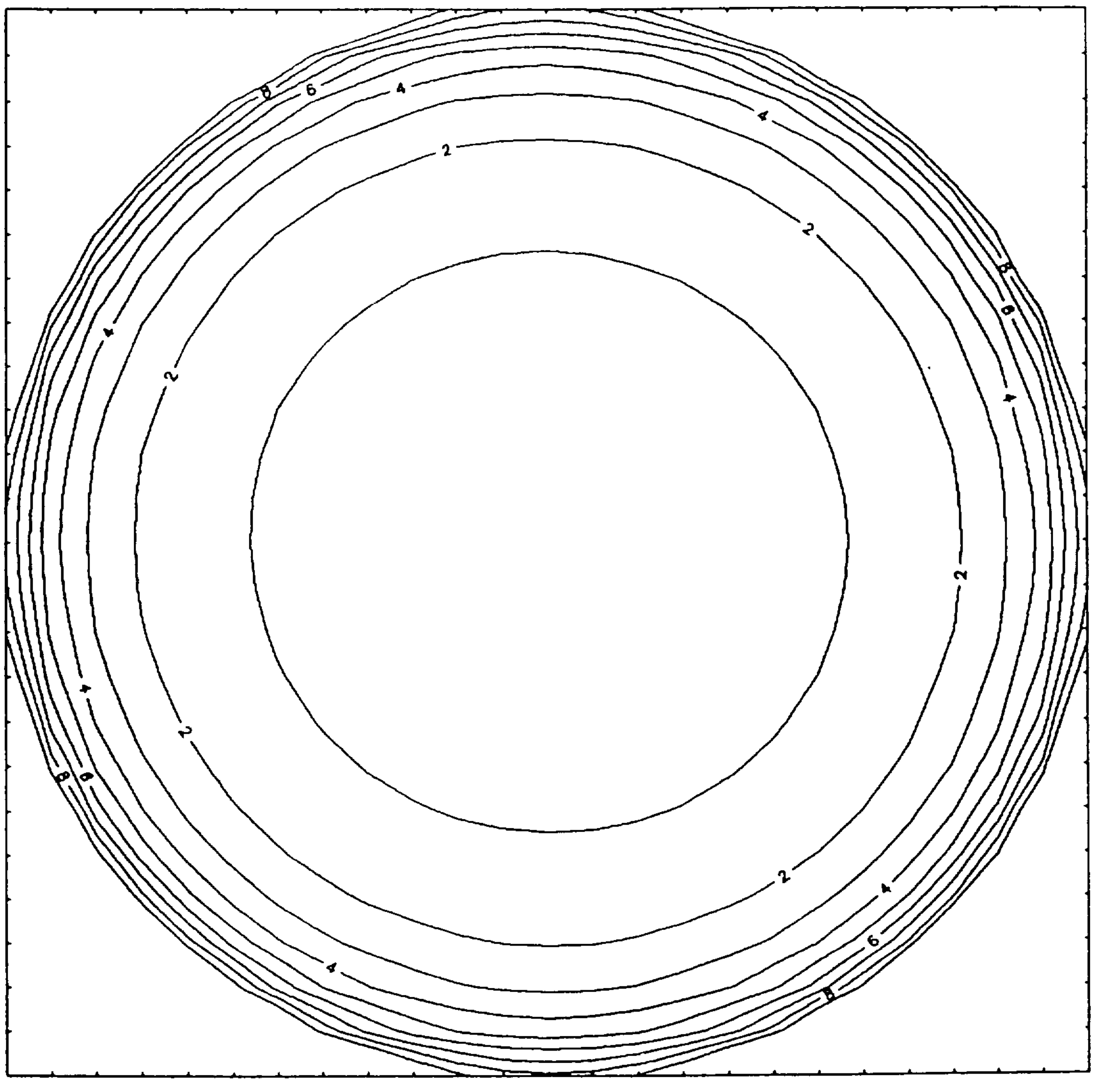


Figure 2.18 Sensitivity of the calculated Γ to noise in the power detectors over the $|\Gamma| \leq 1$ region with a 0.5 dB attenuator between the sampled-line and the device under test.

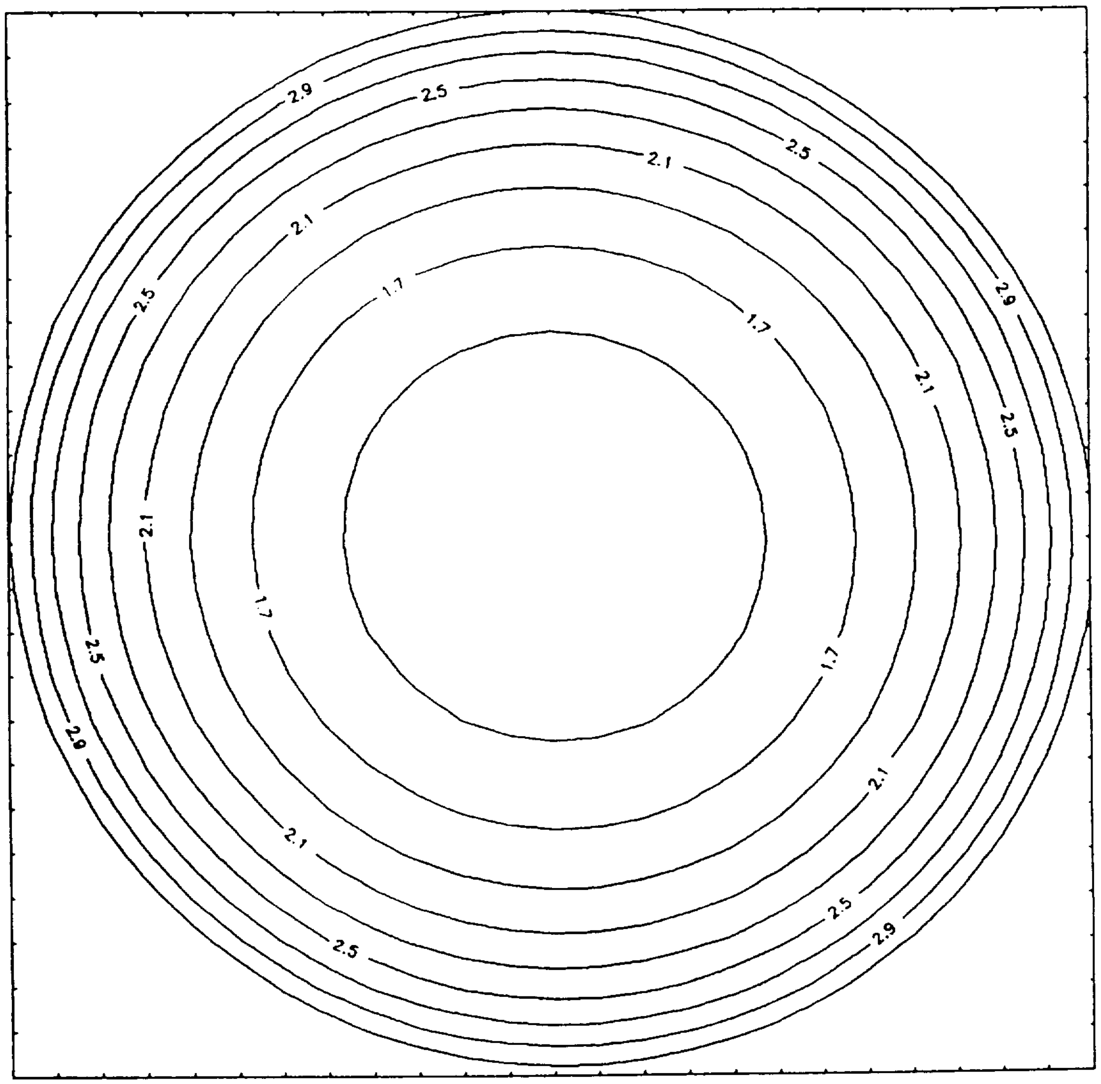


Figure 2.19 Sensitivity of the calculated Γ to noise in the power detectors over the $|\Gamma| \leq 1$ region with a 3 dB attenuator between the sampled-line and the device under test.

An optimum value of attenuator for use with the sampled-line network analyzer when measuring passive loads has been found. It was found by numerically simulating the calibration and measurement sequence of the sampled-line, and finding its worst-case measurement error as a function of frequency. This function was integrated over the octave-wide frequency interval of a particular diode triple. The integrated worst-case error function was then minimized over values of attenuator. The best value was found to be 4.8 dB.

2.3.2 Calibration and Measurement Options with the Sampled-Line

Not all of the calibration options available to the general six-port network analyzer are available to the sampled-line. The most significant of these is the use of a set of arbitrary unknown standards for the six-port-to-four-port phase of the calibration. As was shown in Section 2.2.1, equation (2.55) allows determination of the calibration constants - or iterative improvement of their estimates - through the observation of at least nine arbitrary standards. Unfortunately, the fact that the measurement centers of the sampled-line lie along a line in the complex w-plane makes the resulting matrix singular for the sampled-line.

Calibration of the sampled-line must rely therefore on a sliding short circuit and use of equation 2.59 as described in Section 2.2.1. This is a disadvantage since the quality of the calibration depends then on the quality of the sliding short circuit.

In measurement, the sampled-line offers options not available with the standard six-port. As was noted above, the frequency range of the sampled-line is expanded by adding samplers to the line. There will be n-2 outputs from the sampled-line for a unit covering n octaves. The outputs from the diodes which are not in the primary triple for calibration and measurement can be used in a least-squares optimization to improve the estimate of the reflection coefficient.

2.3.3 Comparison of the Sampled-Line and NBS Six-Ports

Figure 2.20 shows a comparison of the worst case errors for an NBS six port of the type shown in Figure 2.12, and for a sampled-line network analyzer. This figure is based on a numerical simulation in which both network analyzers were calibrated in the same way, with a sliding short and short, open, and load impedance standards. It was assumed that power detectors in both instruments were perfect square-law detectors with a resolution of 50,000 counts. RMS noise of two counts on each detector was assumed. The model of the sampled-line included the effects of line loading by the samplers. Each sampler was modeled as a shunt admittance of $0.1Y_0$ where Y_0 is the characteristic admittance of the sampled line. The model for the NBS analyzer assumed perfect couplers and perfectly matched detectors.

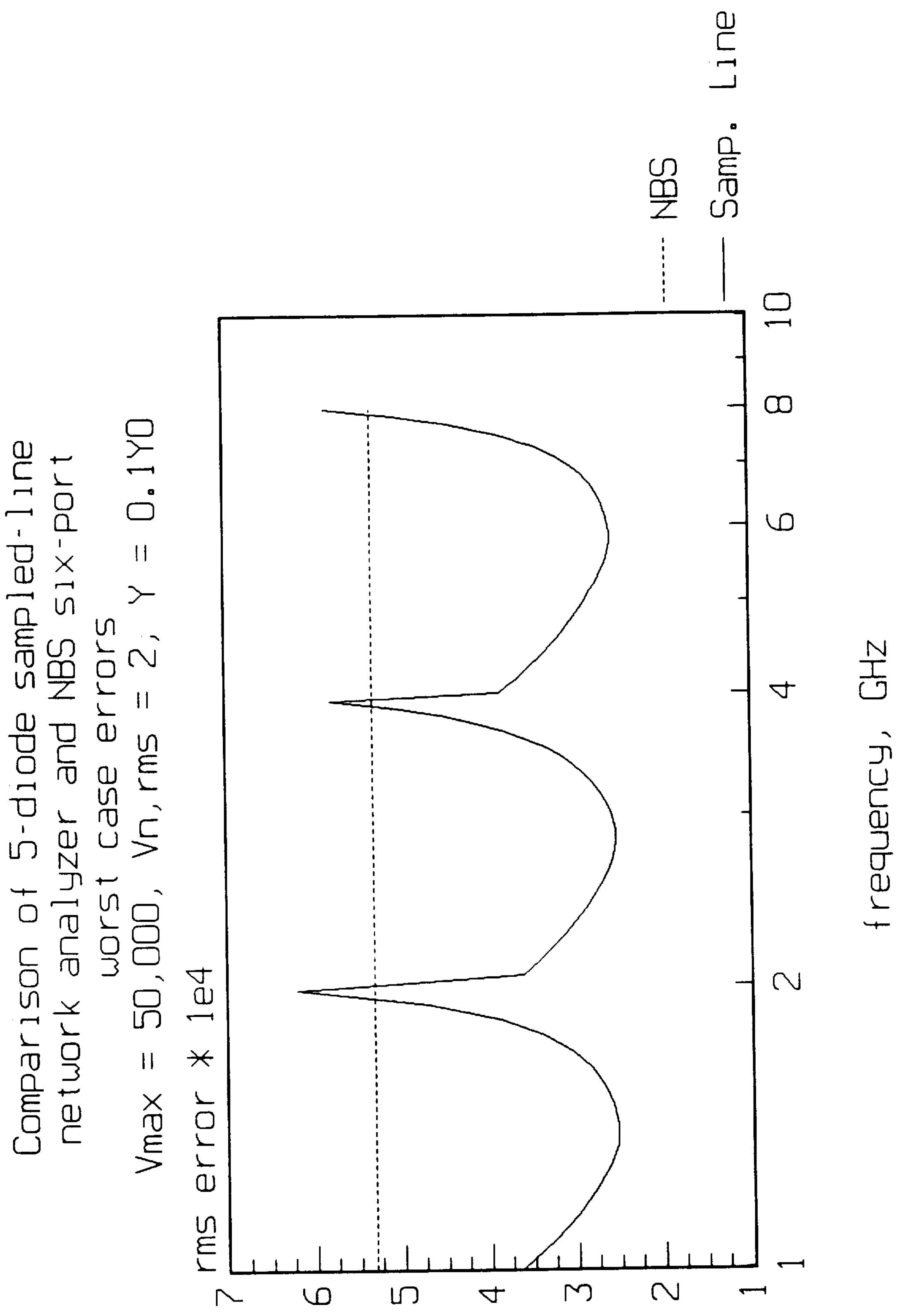


Figure 2.20 Comparison of the sampled-line network analyzer with an NBS-type six-port network analyzer. Ideal models are used for couplers in NBS circuit. Sampled-line is optimized for 1-8 GHz operation and samplers look like 500 ohm resistors loading the line.

We see that whereas the NBS type six-port has a worst case error which is independent of frequency, the sampled-line's error varies with frequency, reaching maxima at the band edges when switching from one diode triple to another. For most of the range however, the sampled-line is superior to the NBS network analyzer in measurement error.

Since the ultimate goal of this work is to operate at very low test signal powers, a comparison of the two devices in terms of the ratio of power delivered to a detector diode to the power delivered to the device under test. For this comparison it was necessary to make an assumption about the matching networks used for the detector diodes in the NBS six-port. We assumed ideal isolators were used, though in general some lossy element, such as an attenuator would need to be used to achieve the broadband match required here. The diode impedance is typically a kilohm or so, shunted by a few tenths of a pF. For the sampled-line, it was assumed that isolation resistors of value $10Z_0$ were placed between the diodes and the line. The same model for the detector diode was used for both.

The result is that the two network analyzers are within a dB of each other in this respect up to the corner frequency of the sampled-line's sampling circuit. The roll-off is caused by the sampling diode's capacitance, and since the diode sees a higher embedding impedance in the sampled-line analyzer than in the NBS, its corner comes at a lower frequency. Thus, in terms of the power delivered to the detectors, the two analyzers are comparable at low frequencies, below about 2 GHz for the model assumed here. Above 2 GHz, the sampled-line loses ground on the NBS-type at the rate of 6 dB/octave until the NBS hits its corner frequency at perhaps 7 GHz. Above this corner frequency, the NBS analyzer has a steady advantage of 8 or 9 dB.

Since diode matching networks in the NBS six-port would in general have losses, it is clear that the the two types are not largely different in power delivered to the detectors. The small size achievable with the sampled-line analyzer thus tips the balance in its favor for this application.

In summary then, the sampled-line has better measurement accuracy over most of any given range of frequencies than the NBS-type analyzer. It is comparable in power delivered to the detector diodes for a given power available to the device under test. Its calibration options are more limited than those of the NBS device, but the large number of detectors used offer the potential of improved accuracy through use of the redundant information.

2.4 Construction of the Sampled-Line Module

The first step in construction of the microwave sampled-line module was the construction and testing of two candidate sampler circuits using different fabrication techniques. The relative merits of these two approaches were to be evaluated, and and one was to be selected for use in construction of the final sampled-line modules.

Schematically, the two sampling circuits considered were the same. The sampler is shown schematically in Figure 2.21. It consists of an isolation resitor and dc blocking capacitor between the diode and the line, and an RC low-pass filter following the detector to isolate the RF from the low-frequency lines. The detector used is a Hewlett-Packard HSCH-5336 beam-leaded Schottky diode. The difference in the two approaches was in the fabrication of the isolation resistors used for the samplers. In one case a 500 Ohm microwave chip resistor was affixed to the microstrip line using conducting epoxy. In the other, an Ohmega-Ply substrate from Rogers, Inc. was used. The Ohmega-Ply material allows planar resistors to be fabricated directly on the board through an etching procedure. It was thought that the Ohmega-Ply material would give lower parasitic loading of the microstrip line than the chip resistors.

Both circuits were built and their microwave scattering parameters were measured over the 0.5-10 GHz range. Over this range, parasitic loading effects for both samplers appear quite small. Calculated scattering parameters from a model consisting of a pure resistance shunting a transmission line agreed with the measured data fairly well. The parasitic shunt capacitance introduced by either sampler appears to be in the tens of femtofarads.

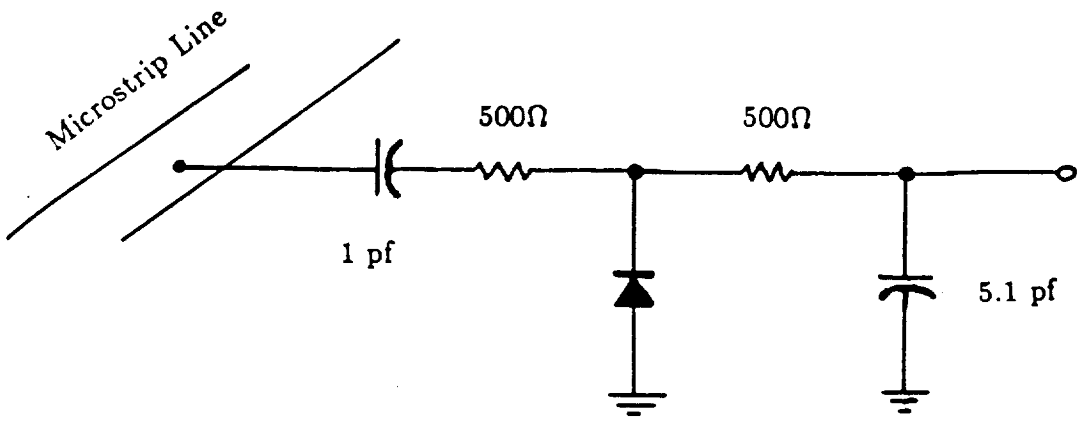


Figure 2.21 Sampling circuit used in the sampled-line network analyzer.

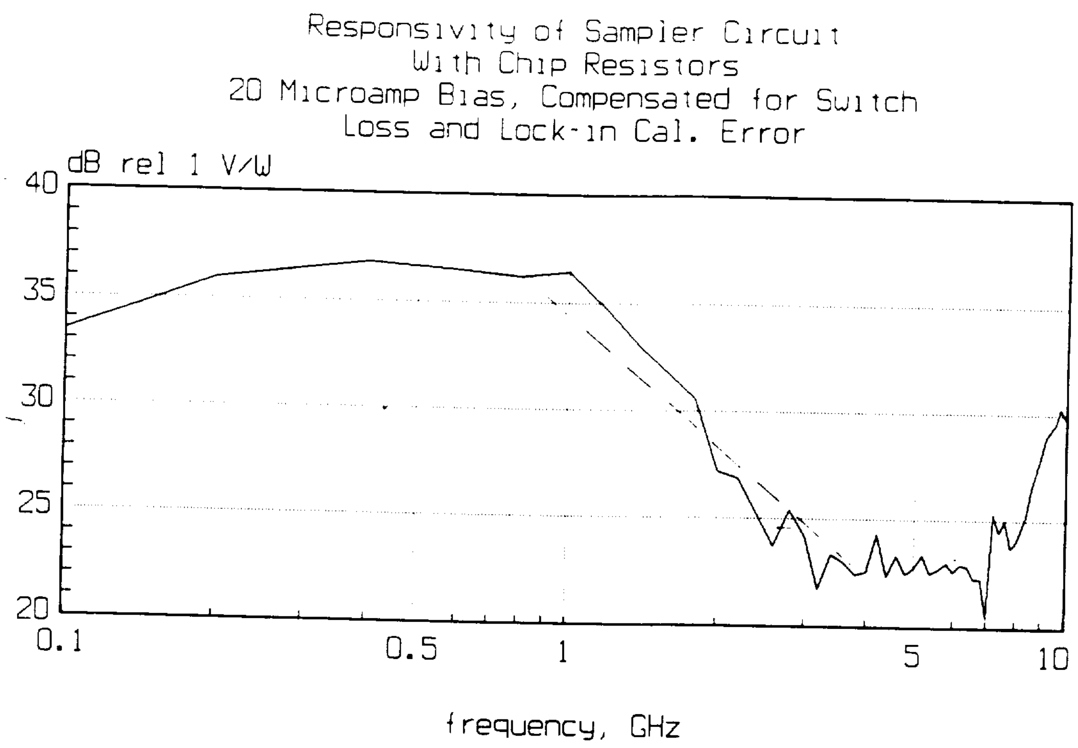
The measured responsivities of the two samplers showed a marked difference in them, however. The results are shown in Figure 2.22. The responsivity of the Ohmega-Ply sampler falls off sharply above 6 GHz, while the chip resistor version performs satisfactorily up to 10 GHz. It is thought that parasitic inductance in the Ohmega-Ply resistor is the cause of this sharp fall. The resistive material on the substrate has a sheet resistivity of Ohms per square. Thus, a 500 Ohm resistor must have an aspect ratio of 20 to 1. The resistor we fabricated was 0.15 mm by 3 mm. Some parasitic inductance in such a long thin line would be expected. A simulation in which the resistor was modeled as a lossy transmission line gave results in good agreement with the measured data.

Since the sampled-line analyzer to be built must perform up to 8 GHz, the chip resistor sampler design appeared to be the one of choice. Tests in JPL's cryogenic dewars showed that the chip resistor design could survive multiple cooling cycles to 4K without ill effects. The chip resistor design was thus chosen for the sampled-line module.

The sampled-line module which is the primary component of the measurement system consists of a microstrip line, connectorized on both ends, in a brass enclosure. Five resistive samplers of the type described above are distributed along the line, spaced logarithmically to give good measurement performance over the 1-8GHz range. Low frequency output signals from the detectors leave the enclosure through a doubly shielded multi-coax connector. An attenuator, fabricated in the microstrip using thick-film resistors, is placed between the sampling diodes and one connector to prevent deep nulls on the sampled line. Further details of the design, and results of various tests are presented in the following paragraphs.

Attenuator Design: As noted in previous sections, an attenuator must be placed between the samplers on the line of the instrument, and the device under test. This attenuator prevents deep standing wave nulls on the sampled-line, which degrade measurement accuracy. In the past, an external coaxial attenuator was used, but due to size constraints on the current application, it was deemed necessary to integrate the attenuator into the sampled-line module itself.

Previous work has shown that the optimum attenuator value for use with the sampled line is 4.8 dB. We designed a simple T- attenuator of this value. The series arms of the attenuator were 13.21 Ω resistors, and the shunt leg was 88.04 Ω. We used a Rogers Ohmega-Ply



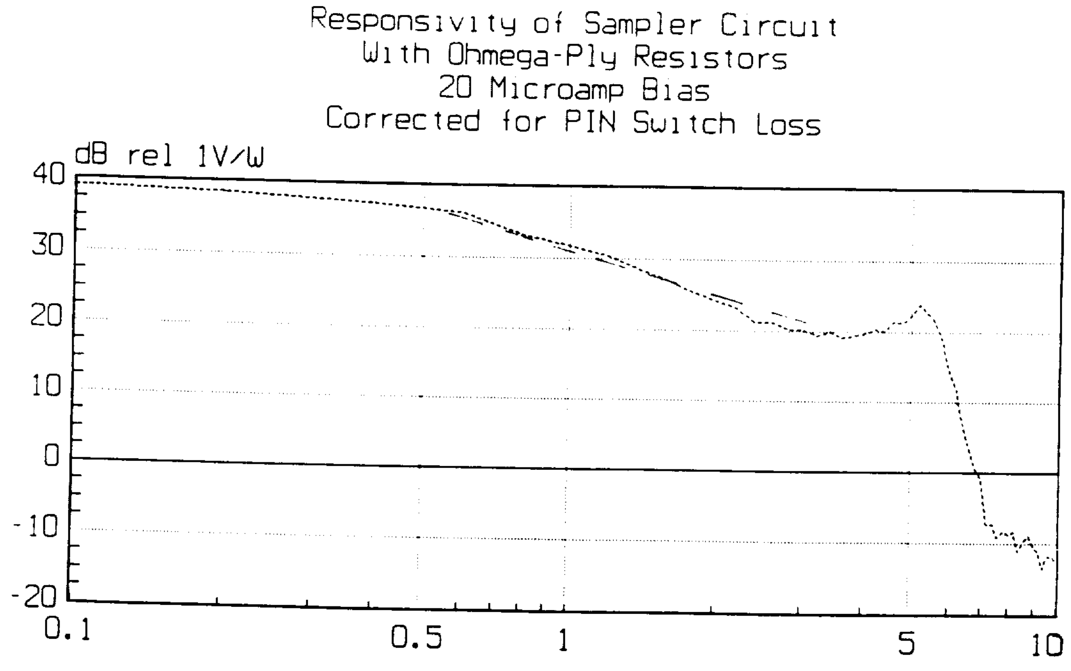


Figure 2.22 Measured responsivities for two sampler circuits. One was built with chip resistors epoxied to the circuit board. The other used integrated resistors fabricated on the board through use of the Rogers Ohmega-Ply substrate material. Left-hand scale is in dB relative to 1 volt per watt travelling down the sampled line. The measurement was made with a matched termination on the microstrip line.

substrate to fabricate this attenuator. The Ohmega-Ply substrate is a copper-clad microwave substrate. Under the copper cladding is a layer which, when exposed and chemically treated, becomes a 25 Ω/square resistive layer. A 50 Ω microstrip line on our substrates is 1.1 mm wide, so our series resistors became 0.6 x 1.1 mm bars, and the shunt resistor was 0.25 mm wide and 0.9 mm long. A via hole through the board supplied a ground connection for the shunt leg.

This attenuator configuration was arrived at after a couple of trials. We found that the shunt resistor must be made as physically small as possible, to move parasitic transmission line effects in the shunt resistor up beyond the frequency range of interest.

Figure 2.23 shows the performance of the microstrip attenuator. Over the 1-8 GHz frequency range, the attenuation is flat to within +/- 0.5 dB, and the return loss is greater than 14 dB. These specs are comparable to those of commercial drop-in chip attenuators, and are adequate for our present purposes.

Sampled-Line Module Construction: The sampled-line module layout incorporated five samplers of the type tested at low temperatures, and a microstrip attenuator. The samplers were spaced in a log-periodic fashion to give good measurement accuracy over a 1-8 GHz frequency range. The microstrip circuit was mounted with conducting epoxy in a machined brass housing.

Sampled-Line Module Testing: Figure 2.24 shows |S11| and |S21| of the sampled-line module. The transmission loss of about 7 dB is accounted for by the internal attenuator, and the effects of the samplers. We calculate that microstrip loss in the module accounts for only 0.1 dB at the low end of the band and 0.6 dB at 10 GHz.

The sampled-line exhibits a return loss of about 15 dB or more over the 1-8 GHz band, which indicates that the resistive samplers are not badly loading the line.

Figure 2.25 shows the responsivities of the five detectors in the sampled-line module. The responses of the detectors are reasonably well matched. Some variance is inevitable since upstream samplers slightly reduce the power available to their downstream neighbors. The responsivities seen here agree well with those observed for a single sampler. We have not found a conclusive explanation for the peaking of the responsivity at 10 GHz, but circuit modelling work indicates that a resonance between the via hole inductance and the Schottky diode capacitance may be the cause.

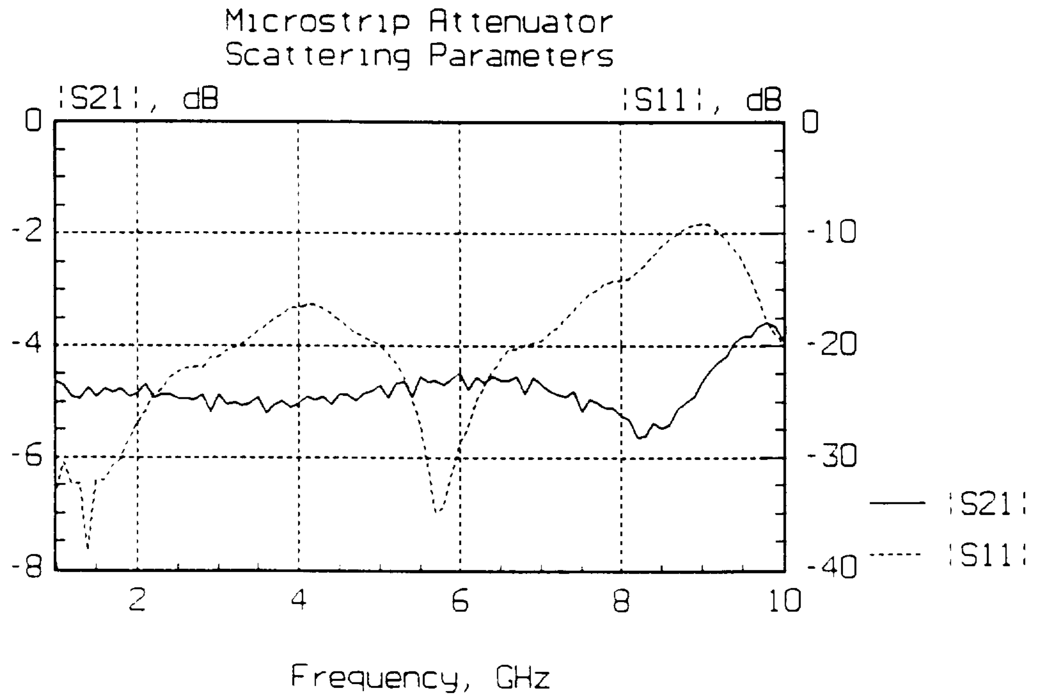


Figure 2.23 The microstrip attenuator integrated into the sampled-line module exhibits attenation flatness of ±0.5 dB over the 1-8 GHz band.

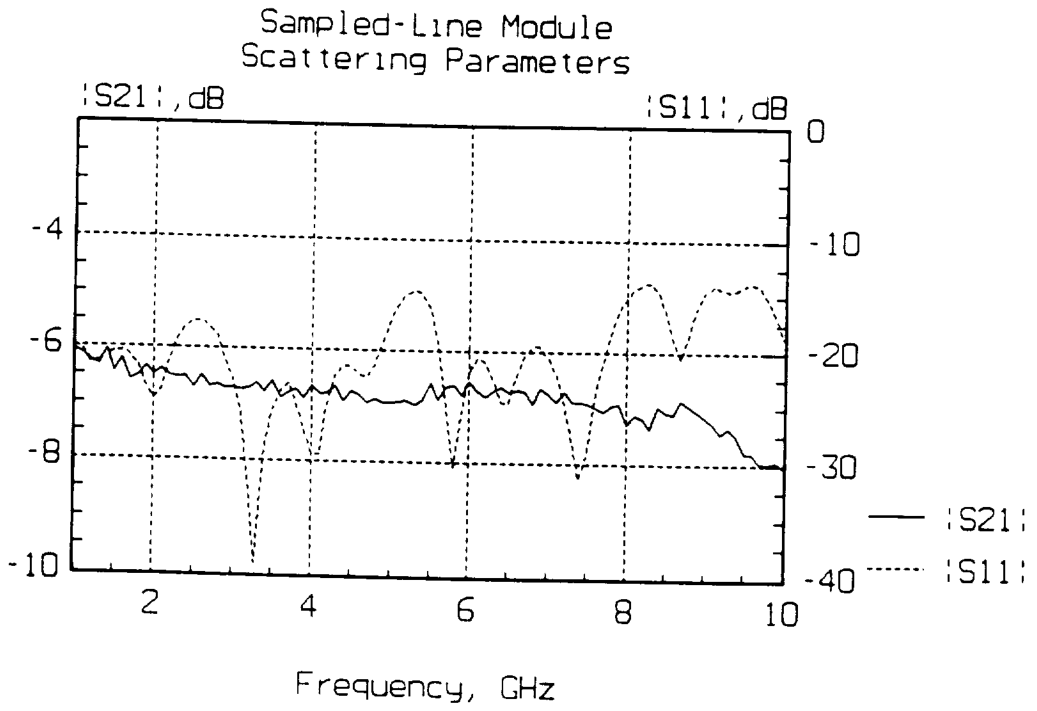


Figure 2.24 S-parameters of the sampled-line network analyzer module.

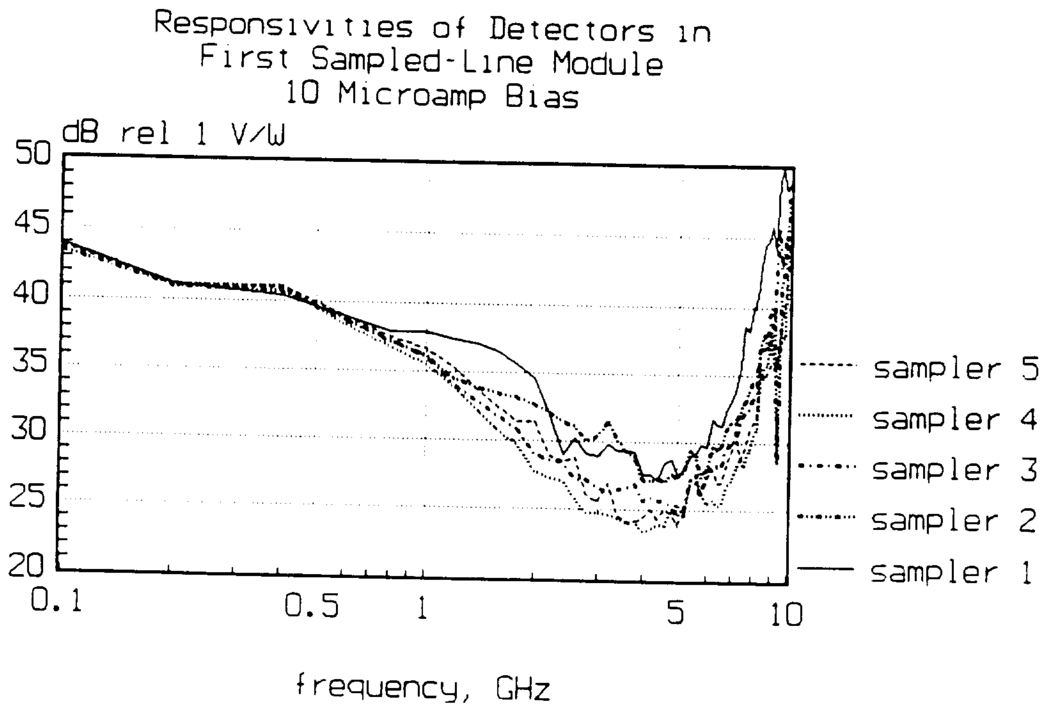


Figure 2.25 Detector responsivities of the sampled-line module.

The sampled-line module was connected to the breadboards of our post-detection signal processing circuitry. With a 300 nW test signal and a 0.1 sec integration time, we observe an SNR of about 45 dB on the final output of our detection circuitry. With a 10 sec integration, the test signal level which would drop the output of this system to the noise floor is about 170 pW.

2.5 Calibration and Measurement Tests with the Sampled-Line Module

The post-detection electronics described in Sections 3 and 4 below were combined with the network analyzer module, and calibration and measurement tests were performed on the resulting network analyzer system at room temperature, 77K, and 4K.

Figure 2.26 shows the results of a typical room temperature test. The network analyzer was calibrated, and the reflection coefficient of a short circuit at the end of a length of precision air coaxial line was measured. This test was performed with a relatively high test signal level. The programmable amplifiers in the post-detection electronics were all set to their minimum gain values, and the test signal applied to the load was a few microwatts.

In the figure, the sampled-line measurement is compared to a measurement of the same device with our computer error-corrected HP 8410 network analyzer. This instrument has an error vector of magnitude 0.01 over this frequency range. We thus estimate that on this measurement, the sampled-line network analyzer had an error of 0.02 for $f < 4$ GHz, and an error of 0.04 for $f > 4$ GHz. We believe this performance can be improved by better characterizing the calibration standards we are now using at the higher frequencies.

Cryogenic tests of the network analyzer were performed at the Jet Propulsion Laboratory using immersion dewars. The procedure was to connect two stainless steel semi-rigid coaxial cables to the network analyzer module, one to each end. These cables were strapped together with the low-frequency output cables from the analyzer to form a boom which could support the analyzer module's weight as it was lowered into the cryogenic bath. With the network analyzer sitting in the bath, the test signal could be applied to input cable, and calibration standards and the device to be measured could be connected to the other cable. The network analyzer was thus calibrated to a reference plane in the "warm" while sitting in the "cold." With this technique, we could perform a variety of tests, make changes to signal level, etc. without waiting through warm-up/cool-down cycles.

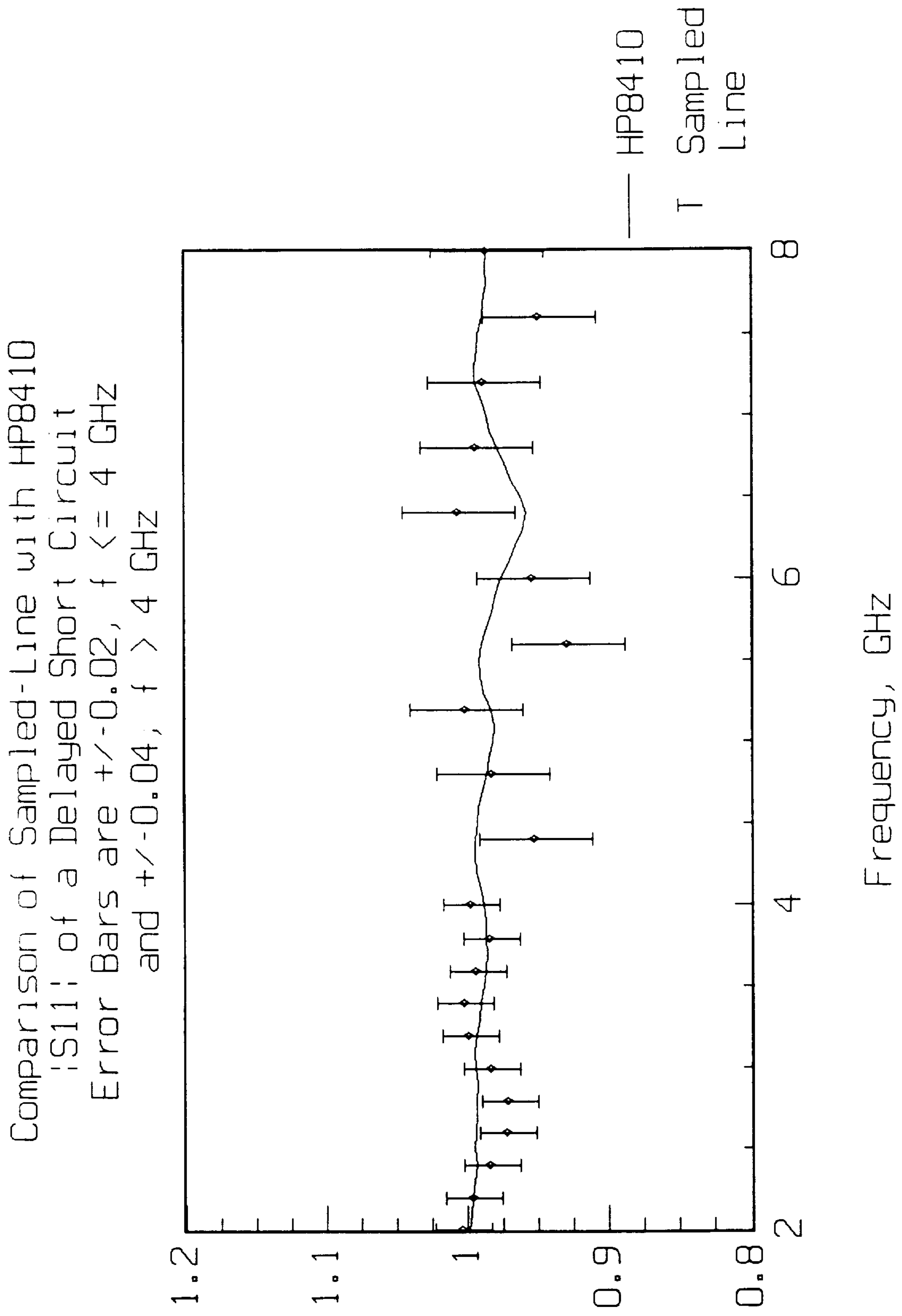


Figure 2.26 Results of a typical room temperature test.

It was found that some changes to our original diode biasing circuitry were required for cryogenic operation. As described below, our original circuitry supported two discrete diode bias currents. The idea was to use one for room temperature measurements, and the other for cryogenic measurements, and to be able to switch between them under software control. Early tests on a single diode sampler indicated that this should work satisfactorily. In our first cryogenic test of the completed sampled-line module, however, it became apparent that with cooling, diode parameters could be quite different from device to device. We thus built an external bias box with independently adjustable bias currents for each diode in the sampled-line module.

The table below shows the results of a measurement of the delayed short circuit with the network analyzer cooled to 77K. The test signal power is a few tens of nanowatts. The test is over the 2-4GHz range, and with the exception of one bad point, the analyzer shows good accuracy. The "Ideal" column shows the reflection coefficient expected for a lossless shorted delay line of the length used. The fact that the phase error increases approximately linearly with frequency may indicate a frequency error in the signal generator.

	Measured		Ideal	
Frequency	**Mag**	**Angle**	**Mag**	**Angle**
2.00000	0.99203	57.0	1.0	60.0
2.20000	0.99266	9.5	1.0	12.0
2.40000	0.98820	-40.2	1.0	-36.0
2.60000	0.96887	-88.5	1.0	-84.0
2.80000	0.97855	-138.0	1.0	-132.0
3.00000	0.98442	174.4	1.0	180.0
3.20000	0.98001	125.1	1.0	132.0
3.40000	0.99239	76.9	1.0	84.0
3.60000	0.99232	27.4	1.0	36.0
3.80000	0.98641	-21.8	1.0	-12.0
4.00000	1.29512	-70.0	1.0	-60.0

A bias current of about 10 microamps per diode works well at both 77K and room temperature. When the analyzer was immersed in liquid helium, however, the detectors essentially ceased functioning. It was found that increasing the bias current to 100 microamps brought the detectors back, and allowed measurements to be made at 4K. The theory does not indicate that these detector should work at all at 4K, so our conjecture is that the increased bias current causes internal heating of the device so that some part of the active region gets warm enough to operate.

The disadvantage of this high bias current is that it increases the diode's diffusion capacitance, and thus reduces the high frequency response of the network analyzer. In our tests to date we have not been able to successfully calibrate the network analyzer at frequencies above 3GHz. The table below shows results for a 4K measurement from 2-3GHz. The bias current was 100 microamps on each diode, and the test signal power was a few tens of nanowatts.

	Measured		Ideal	
Frequency	Mag	Angle	Mag	Angle
2.00000	0.97244	58.6	1.0	60.0
2.20000	0.88273	7.8	1.0	12.0
2.40000	1.04530	-39.1	1.0	-36.0
2.60000	1.11285	-87.2	1.0	-84.0
2.80000	1.05116	-140.6	1.0	-132.01
3.00000	0.94307	172.8	1.0	180.0

3.0 POST-DETECTION ELECTRONICS DESIGN AND CONSTRUCTION

Due to the extremely low-level signals to be detected as outputs from the sampled-line module, the network analyzer system requires high-performance post-detection amplifiers, and a synchronous detection scheme for signal recovery. An analog-to-digital converter system is also required to enter the analog outputs from the detectors into the computer for processing.

After some experimentation, the architecture shown in Figure 3.1 was chosen for the instrumentation electronics. It provides good performance and flexibility of configuration, to allow measurements in several regimes of signal power level.

The system essentially consists of a multi-channel PC-based lock-in amplifier. The synchronizer output signal is applied to a PIN-diode modulator, giving a square-wave modulated microwave test signal. The outputs from the analyzer's detectors are thus similarly modulated.

This square-wave modulated detector output is amplified through a series of AC-coupled amplifiers and applied to a balanced demodulator. The balanced demodulator's LO input is a square wave of the same frequency and phase as the detected signal. Thus, only the detector output gives rise to a DC component in the output of the demodulator. The cutoff frequency of the low-pass filter following the demodulator can be adjusted to give very low overall system noise bandwidths, and thus high sensitivity.

The flexibility of the system lies in the programmable gain amplifiers and the programmable cutoff frequency filter. For measurements with very low signal powers, the system gain can be maximized and the LPF cutoff minimized to give the best sensitivity at the expense of measurement speed. When a higher test signal power may be used, turning down the gain and opening up the LPF gives a faster measurement system.

The outputs from the low-pass filters go through another gain stage, and then are multiplexed into a 16-bit A/D converter. The A/D converter exchanges data with the computer (an IBM AT) over a simple digital interface based on an Intel 8255 programmable peripheral interface.

We designed the post-detection electronics package around VMEbus hardware. The system was designed to be expandable, with one central MUX, A/D, and computer interface, and with up to 16 channels of lock-in amplification installable as plug-in cards. The lock-in amplifier plug-in cards each contained everything in the signal processing chain of Figure 3.1 from the low-noise amplifier at far left to the programmable integrator.

Each lock-in amplifier is placed on a single 3U by 220 VMEbus size circuit card. The computer interface and power supply are on 6U by 220 VMEbus size cards. Interface to the computer is through a 24-bit parallel general purpose interface card. The entire system resides in a 12"x14"x18" cabinet which can be adapted for mounting in a standard 19" rack.

Due to the low-level signals to be used in this work, each channel of the amplifier must have high gain and low noise. Referring again to Figure 3.1, we will step through the functional blocks of a channel of the multi-channel amplifier and discuss its features and performance.

The low-noise ac-coupled first stage of each amplifier determines the amplifier's noise performance. The noise contributions of subsequent stages are effectively divided by the gain of the first stage. Our first stage is built with three operational amplifiers. The first in line is a Linear Technologies LT1007 low-noise precision op-amp. The gain of the first stage is 86 dB. Its noise performance, measured at a frequency of 1 kHz, is shown in Figure 3.2. For low source impedances, the amplifier has an equivalent input voltage noise density of 6 $nV/\sqrt{Hz}$. At 2 kΩ, typical of the source impedances the amplifiers see looking into the network analyzer module, the noise is approximately 7.5 $nV/\sqrt{Hz}$. At higher source impedances, the noise increases sharply. This is due to increased thermal (Johnson) noise from the source resistor itself, and due to op amp noise currents flowing in the source resistor.

The solid line in Figure 3.2 shows the calculated noise performance for the amplifier. The calculations used the manufacturer's worst case numbers for equivalent input voltage and current noise, and included the Johnson noise of all the resistors in the first stage. For low source impedances, our measured performance is slightly better than that predicted by the calculation. In this regime the noise performance is determined by the op amp's equivalent input noise voltage (shot noise). For high source impedances, our measured noise is slightly higher than the predicted value. This would suggest that the LT1007's input current noise is higher than that quoted by the manufacturer. We have identified some circuit configuration and resistor value changes which may further reduce the equivalent input noise for source impedances in the 2 kΩ range. These changes are currently being implemented.

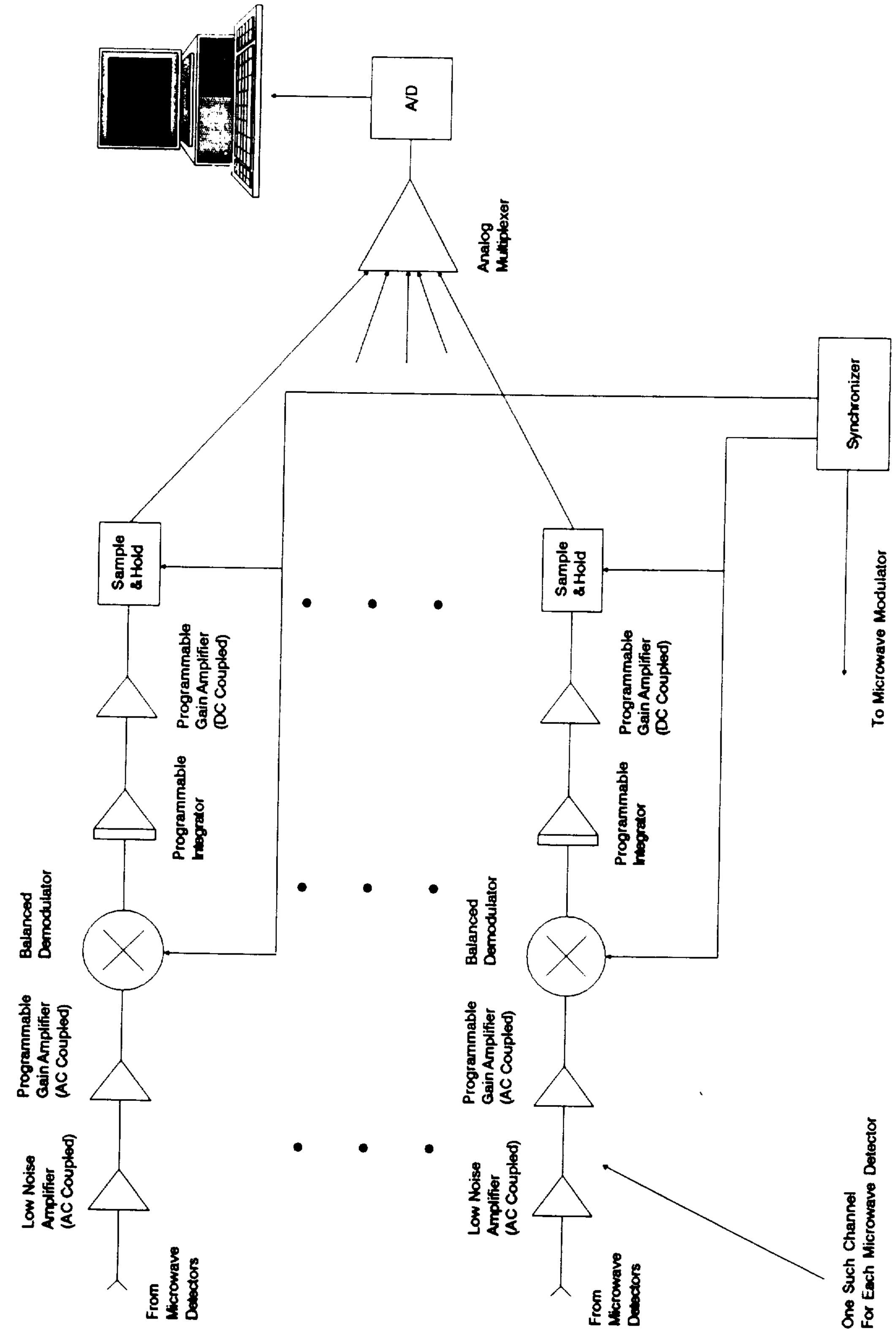

A/D
Analog Multiplexer
Synchronizer
To Microwave Modulator
Sample & Hold
Programmable Gain Amplifier (DC Coupled)
Programmable Integrator
Balanced Demodulator
Programmable Gain Amplifier (AC Coupled)
Low Noise Amplifier (AC Coupled)
From Microwave Detectors
Sample & Hold
Programmable Gain Amplifier (DC Coupled)
Programmable Integrator
Balanced Demodulator
Programmable Gain Amplifier (AC Coupled)
Low Noise Amplifier (AC Coupled)
From Microwave Detectors
One Such Channel For Each Microwave Detector

Figure 3.1 Instrumentation electronics for the sampled-line network analyzer consist of a multi-channel PC-based lock-in amplifier.

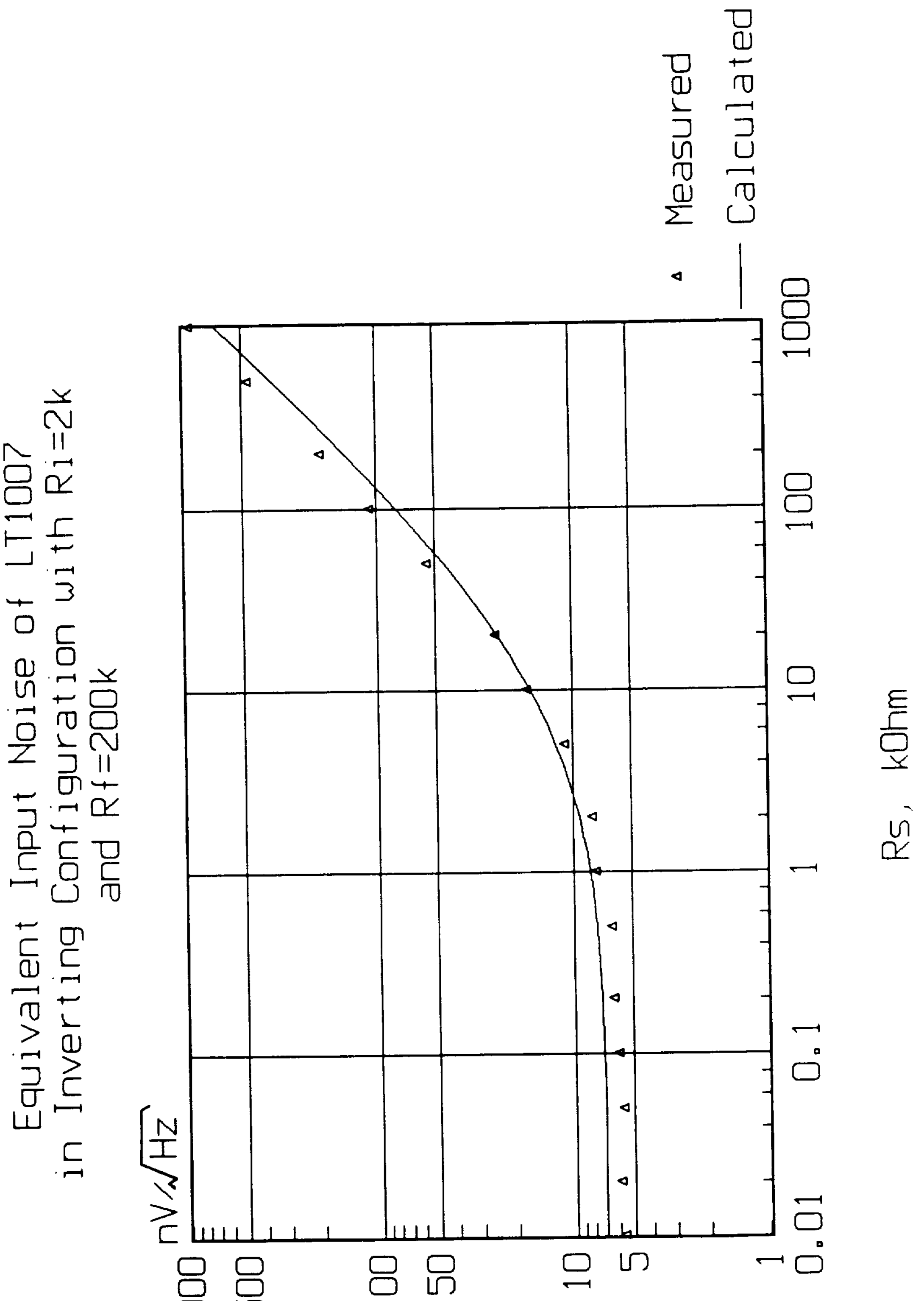


Figure 3.2 Noise performance of a typical channel of the post-detection electronics.

The next block in the amplifier chain is the programmable-gain ac amplifier. This amplifier is built around a general purpose Precision Monolithics OP-270 op amp and its voltage gain is selectable with values of 1, 10, or 100.

The balanced demodulator is a single IC. We use the Analog Devices AD-630. The programmable integrator and programmable dc amplifier are both built around Precision Monolithics ultra-low offset OP-177 op amps. The integrator, a two-pole low-pass filter, can be set for integration times of 0.01, 0.1, or 1 sec. The dc amplifier can be set for voltage gains of 1, 2, 5, or 10. Thus the overall gain of the system can be programmed between 86 and 146 dB in approximately 6 dB steps.

Precision Monolithics SMP-10s are used for the sample-and-hold amplifiers, and the analog multiplexers are Analog Devices ADG-528s. A 16-bit A/D converter is used. The converter is a hybrid module, the Burr-Brown ADC71KG. The programmable amplifiers use TTL-compatible CMOS analog switches, and programming is through a set of general purpose latches and registers on each amplifier board. The computer interface is through a 24-bit parallel interface card (Keithley MetraByte PIO-12) which plugs into the IBM AT backplane. The interface is divided into three 8-bit buses for address, data, and control.

Appendix A gives detailed schematics of the signal processing system. Sheet 1 shows the circuitry on each plug-in amplifier card. U1, U2 and U3A make up the low-noise amplifier. U3B and the associated CMOS switches make up the programmable gain amplifier. The balanced demodulator is implemented by U5, and its output goes to a programmable integrator and programmable gain DC-coupled amplifier (U6 and U7 respectively). The other ICs on the board are digital registers for storing control bits.

Sheet 2 of Appendix A shows the sample-and-hold amplifier bank. All input signals are simultaneously sampled to avoid timing errors in sampling. The outputs of the S/H amplifiers go to analog multiplexers U18 and U19. The outputs of the MUXs are buffered and passed on to the 16-bit A/D converters of sheet 3. Sheet 3 also contains address decoding logic and data registers for the A/D outputs. Sheet 4 shows the system power supply.

4.0 SYNCHRONIZATION CIRCUITRY

The synchronizer, shown as a block in Figure 3.1, provides signals which modulate the microwave test signal, and synchronously demodulate it to yield the output signals. In principal, the frequency of these sync signals could be anything, so long as it lies within the pass bands of the ac amplifier chain. We have chosen, however, to use a phase-locked multiplier to generate a sync signal which is a harmonic of the 60 Hz line frequency (960 Hz). Choosing the sync frequency in this way gives the system the maximum rejection of 60 Hz pickup.

Sheet 5 of Appendix A shows the synchronization circuitry. A 60-Hz signal from a low-voltage transformer winding is applied to comparator U39. A CMOS digital phase detector is used. This type of detector also acts as frequency discriminator when the VCO and input frequencies are vastly different, thus assuring pull-in independent of initial conditions. The loop filter U43 feeds the VCO. The VCO's output is divided down by a factor of 16 by U45 and fed back to the phase detector.

5.0 SYSTEM SIGNAL GENERATOR

The signal generator for use with the sampled-line network analyzer must cover the frequency range of 1 to 8 GHz. One must be able to chop the output of the generator with an external synchronization signal. A leveling loop is unnecessary in the generator as the analyzer forms its own loop by reading the reflectometers' diode outputs and feeding back through the computer.

The signal generator must also have a fairly pure signal. This is due to the fact that the network analyzer, using a video detection scheme as it does, cannot discriminate between a signal and its harmonics. In any measurement, the harmonics set up standing wave patterns on the line just as the fundamental does and the detectors read the sum of these. This gives a source of error which cannot be calibrated or averaged out with existing algorithms.

To assure that the harmonics cannot be a problem in a measurement, the harmonics are required to be below the fundamental by more than the dynamic range of the A/D converter. Then errors due to harmonics will be less than 1 LSB of the converter. Thus, the harmonics must be at the -36 dBc level for the 12-bit A/D and at the -48 dBc level for the 16-bit. The generator was designed for the -48 dBc level.

Figure 5.1 shows a block diagram of the signal generator to be delivered. The 1-8 GHz frequency range is covered by an Avantek HTO-1000 hyperabrupt varactor-tuned oscillator over 1-2 GHz and an Avantek AV-7028 YIG-tuned oscillator over the 2-8 GHz range. A coaxial switch selects between the two oscillators and passes the output to an Avantek AFP-21851 YIG-tuned filter. The pad between the oscillators and the filter is to prevent the oscillators from exceeding the power-handling capability of the filter. The pad also prevents significant frequency pulling of the VCO by changes in load impedance.

The center frequencies of both oscillators and the filter are digitally controlled by the system's control. An automatic alignment routine sweeps through the signal generator's frequency range, finds the optimum filter setting for a given oscillator setting, and stores the result in a disk file which is used by other measurement and control programs.

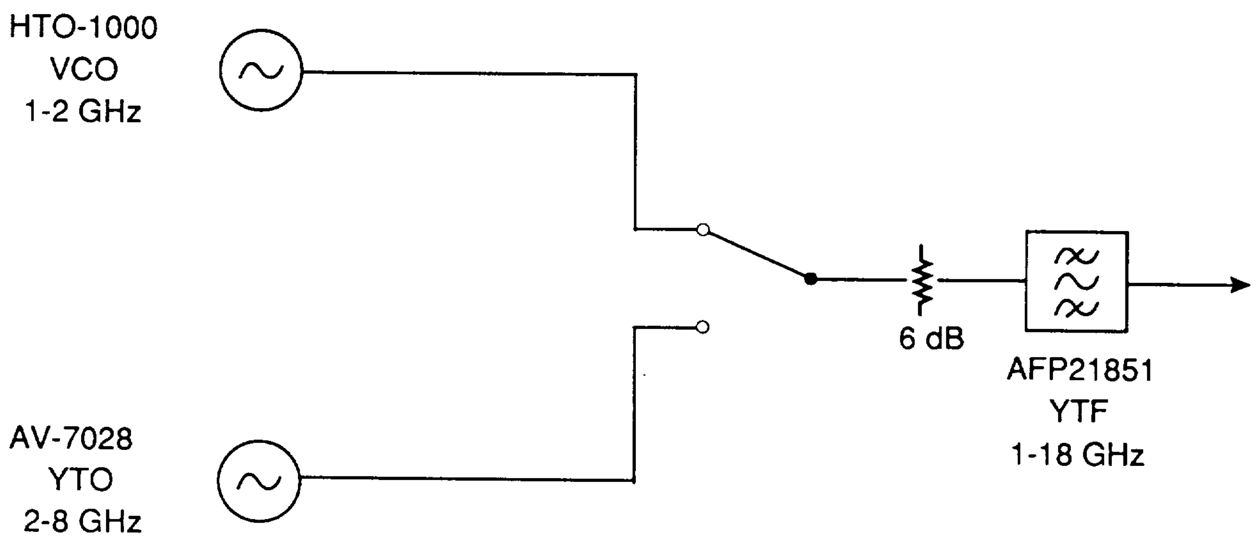


Figure 5.1 Block diagram of the high-purity 2-8 GHz signal generator.

A series of tests were performed on the signal generator to be used with the network analyzer. The results are shown in Figures 5.2-4. Figures 5.2 and 5.3 show spectra of the generator's output signal. The generator uses a hyperabrupt varactor-tuned oscillator to cover the 1-2 GHz range, and a YIG-tuned oscillator for 2-8 GHz. Figure 5.2 shows a spectrum of the varactor-tuned oscillator's output at 1.5 GHz and Figure 5.3 shows the YIG oscillator's output at 6 GHz. In both cases, a slow sweep was used on the spectrum analyzer so the extent of frequency jitter could be observed.

During both measurements, the controlling computer was continuously rewriting the same frequency commands to the signal generator. Thus, the digital interface was active, and the D/A converters were continuously updating the control voltage or current to the oscillator. We consider this our worst case condition from a noise and frequency jitter standpoint. Under these conditions the VCO exhibits about 250 kHz peak-to-peak FM noise, and the YTO FM noise is about 200 kHz peak-to-peak. For the VCO, the peak-to-peak FM noise level corresponds to approximately 1 LSB of the D/A converter driving it. For the YIG, the noise is less than 1 LSB. This level of FM noise should be acceptable for the network analyzer. 250 kHz peak-to-peak of FM noise would correspond to a ±0.15° phase error in measuring transmission through a 1 m delay line.

The output signal was also checked for harmonics and spurious signals. A search with the spectrum analyzer found no spurious or harmonic signals that were larger than -50 dBc

Figure 5.4 shows the output power of the signal generator as a function of output frequency. The discontinuity at 2 GHz is due to the different power output levels of the VCO and YTO.

Sheets 6 and 7 of Appendix A show the digital interface circuitry and analog driver circuitry used in the signal generator.

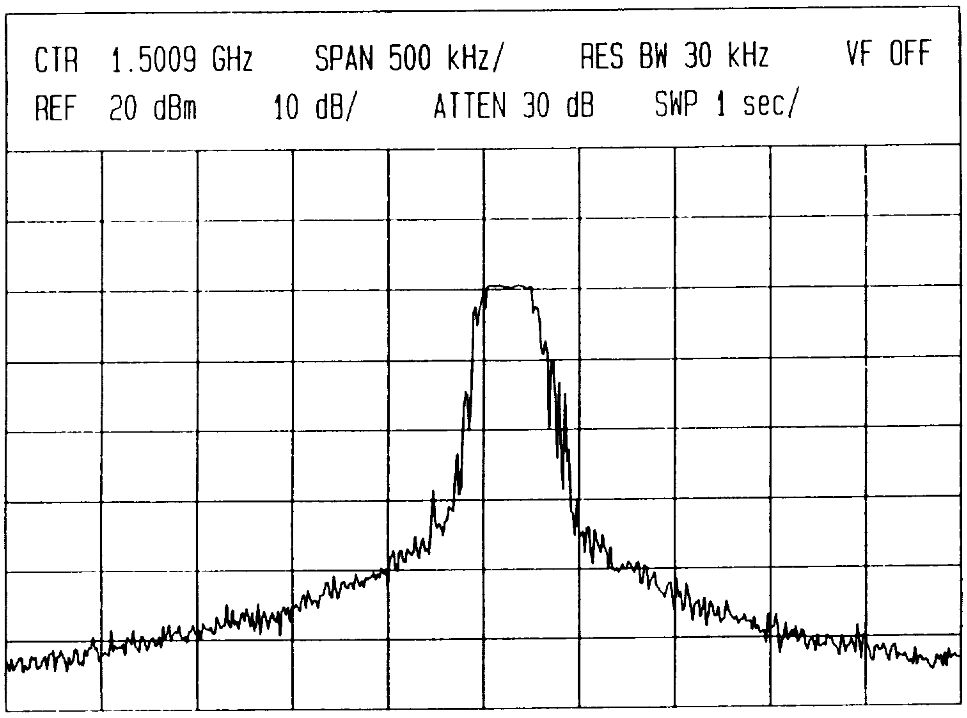


Figure 5.2 The signal generator output at 1.5 GHz shows typical performance of the hyperabrupt varactor-tuned oscillator.

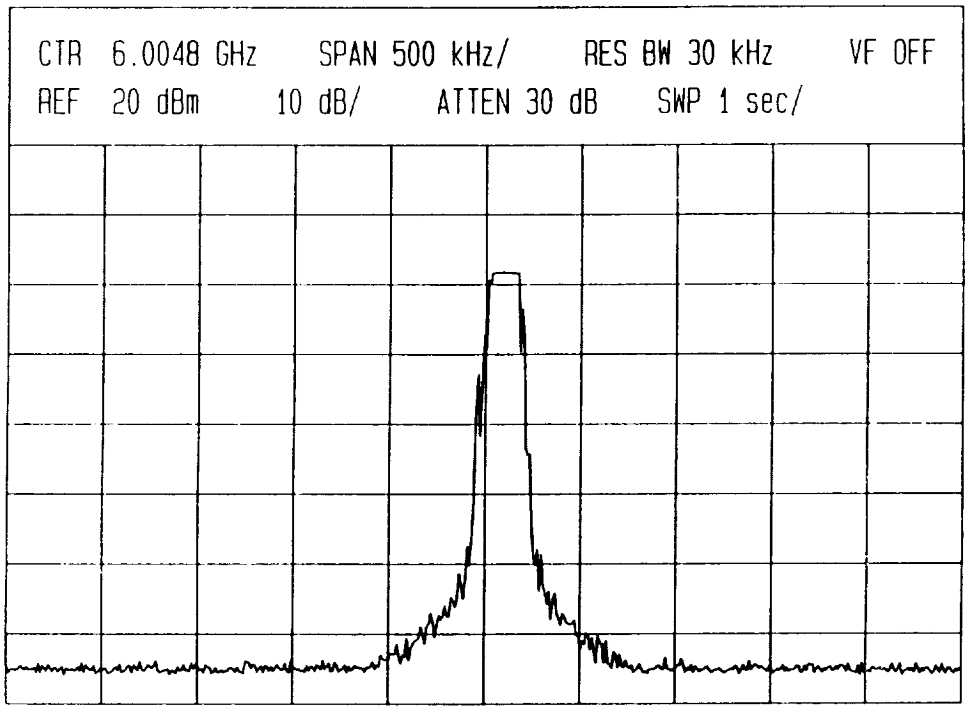


Figure 5.3 The signal generator output at 6 GHz shows typical performance of the YIG-tuned oscillator.

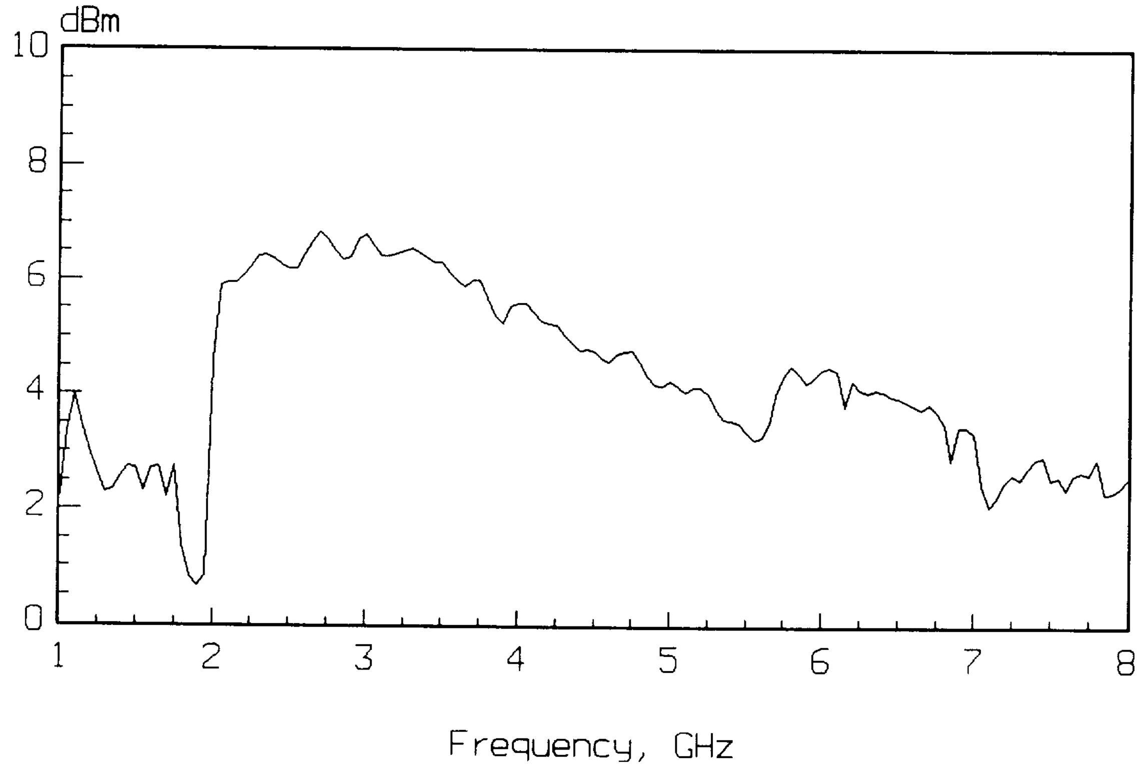


Figure 5.4 Power output of the signal generator was measured using an HP 432A power meter with a HP 478A thermistor mount.

6.0 SYSTEM PHASE SHIFTER

As indicated in Section 2.2, a phase shifter is required for transmission measurements using two six-port network analyzers. Due to the requirements of high repeatability for this instrumentation application, our analyses indicate that a phase shifter using precision mechanical switches and low-loss coaxial delay lines would be required.

A standard digital phase shifter configuration, as shown Figure 6.1, is recommended. To assure well-distributed phases over a three-octave bandwidth a fairly high-resolution phase shifter, perhaps one with six bits would be required. Since the main emphasis in this work has been to get the required performance in reflectometer mode to measure SIS impedances, the phase shifter has not been implemented as part of the deliverable.

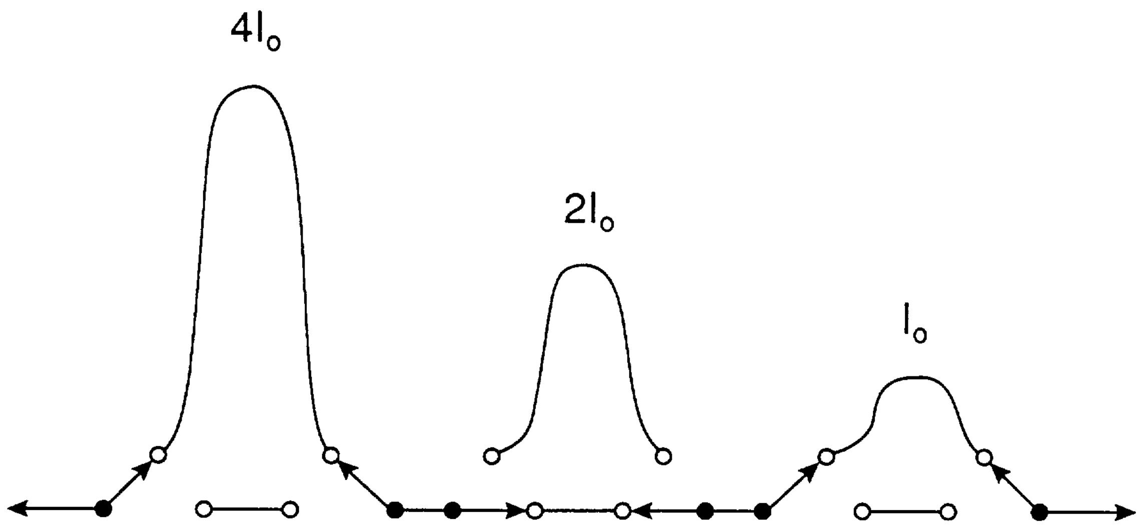


Figure 6.1 A digital phase shifter.

7.0 CONCLUSION

A sampled-line network analyzer which shows promise for characterization of SIS mixers has been built and delivered to workers at NASA/JPL. It is hoped that this unit will enhance that group's capabilities and allow them to make measurements not previously possible.

REFERENCES

[1] R.D. Middlebrook. EE 114, Electronic Circuit Design, Course Notes, California Institute of Technology, 1984-85 academic year.

[2] A.A.M. Saleh. "Explicit formulas for error correction in microwave measuring sets with switching-dependent port mismatches," *IEEE Trans. Instrum. Meas.*, vol.IM-28, no.1, Mar. 1979.

[3] G.F. Engen. "Calibration of an arbitrary six-port junction for measurement of active and passive circuit parameters," *IEEE Trans. Inst. Meas.*, vol.IM-22, no.4, pp.295-299, Dec. 1973.

[4] G.F. Engen. "The six-port reflectometer: An alternative network analyzer," in 1977 *IEEE Mtt-S Int. Microwave Symp. Dig.*, June 1977, pp.44-45, 53-55.

[5] G.F. Engen. "Calibrating the six-port reflectometer by means of sliding terminations," *IEEE Trans. Microwave Theory Tech.*, vol.MTT-26, no.12, pp.951-957, Dec. 1978.

[6] *CRC Standard Math Tables*, Boca Raton, FL: CRC Press, Inc.

[7] G.F. Engen. "An improved circuit for implementing the six-port technique of microwave measurements," *IEEE Trans. Microwave Theory Tech.*, vol.MTT-25, no.12, pp.1080-1083, Dec. 1977.

[8] G.F. Engen and C.A. Hoer. "'Thru-reflect-line': An improved technique for calibrating the dual six-port automatic network analyzer," *IEEE Trans. Microwave Theory Tech.*, vol.MTT-27, no.12, pp.987-993, Dec. 1979.

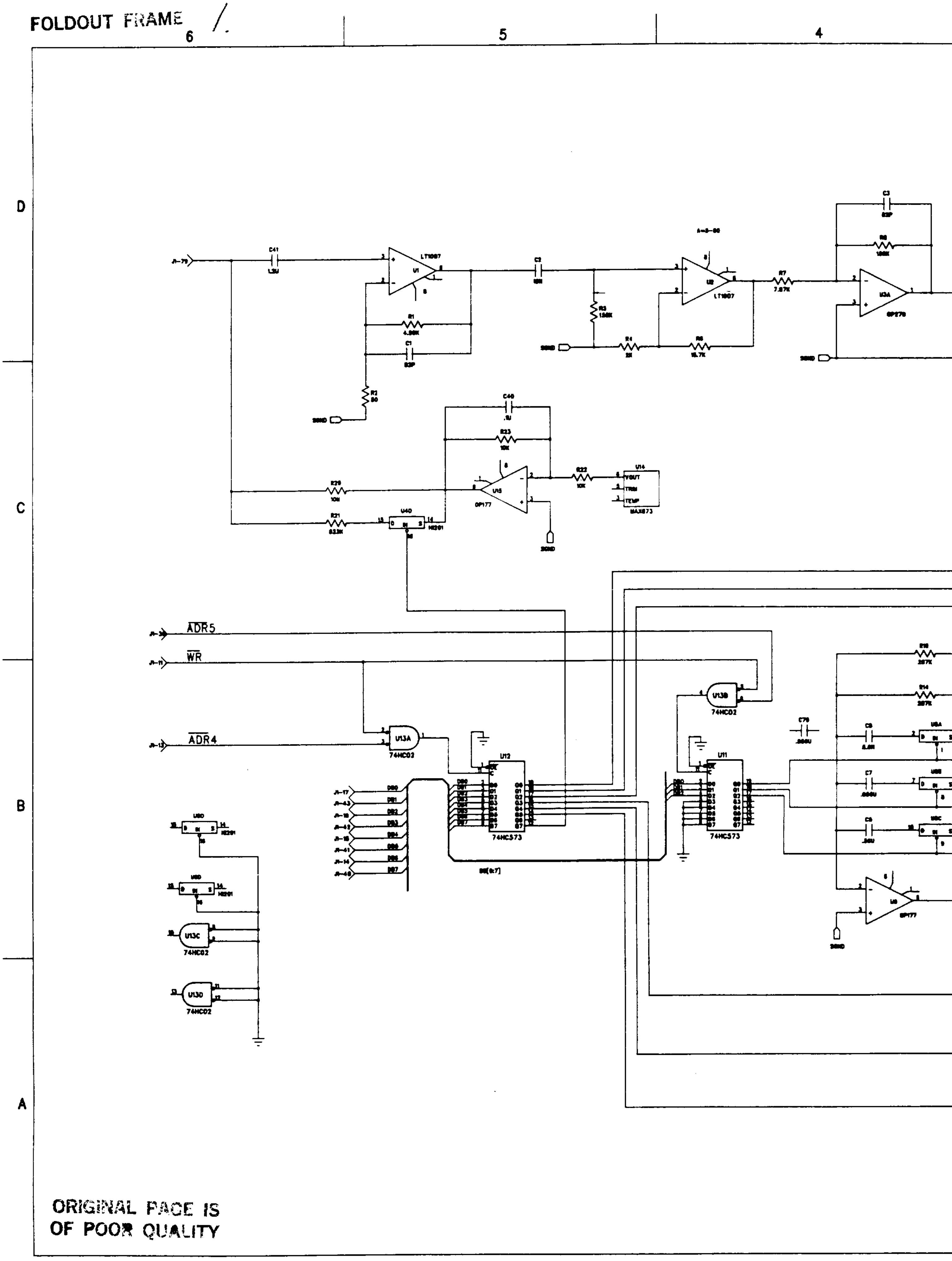
[9] C.A. Hoer. "A network analyzer incorporating two six-port reflectometers," *IEEE Trans. Microwave Theory Tech.*, vol.MTT-25, no.12, pp.1070-1074, Dec. 1977.

Dynamics Technology, Inc.

DTW-8944-91003

APPENDIX A

FOLDOUT FRAME /



ORIGINAL PAGE IS
OF POOR QUALITY

FOLDOUT FRAME 2

3	2	1

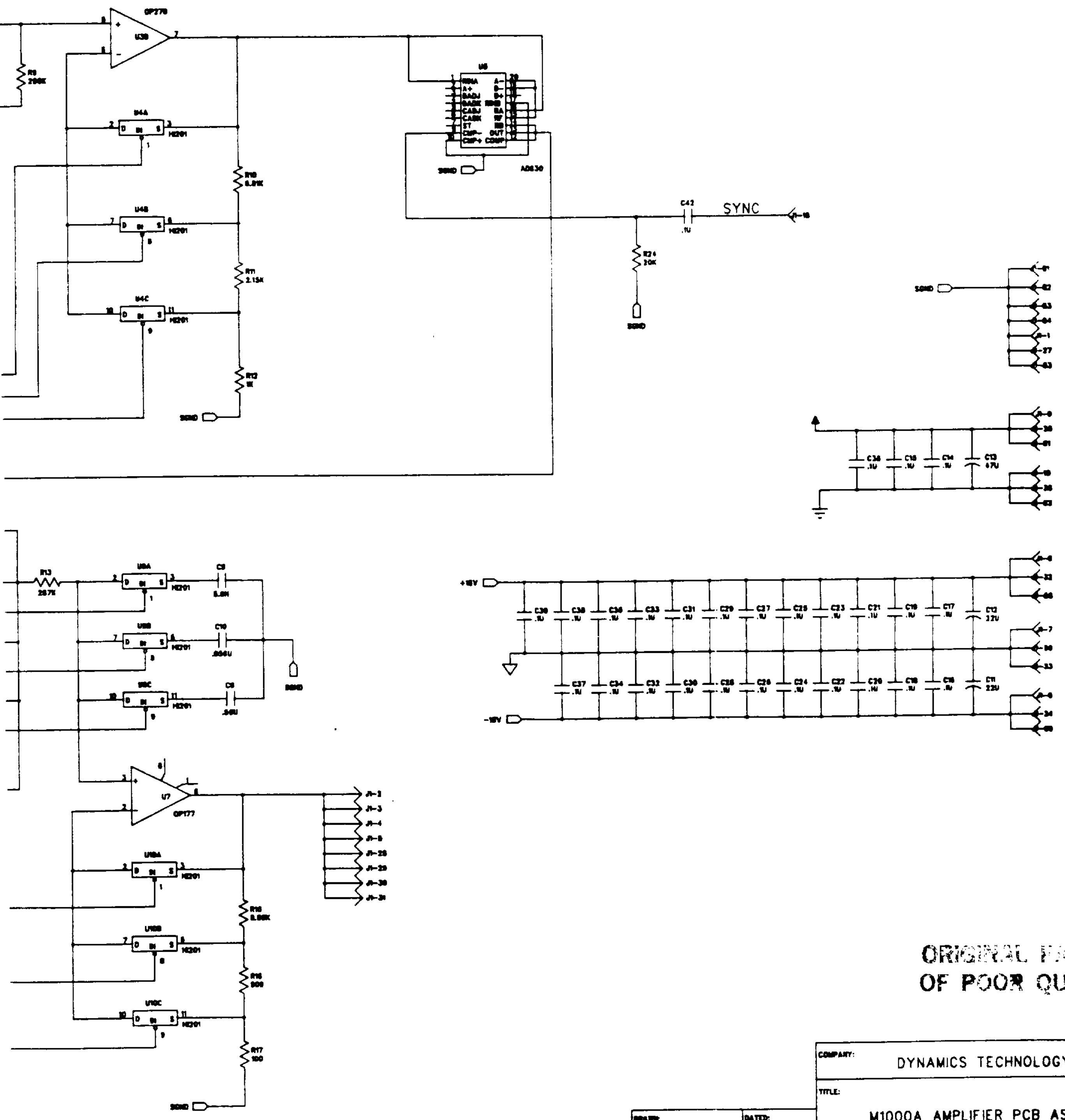
REVISION RECORD			
LTR	ECO NO:	APPROVED:	DATE

D

C

B

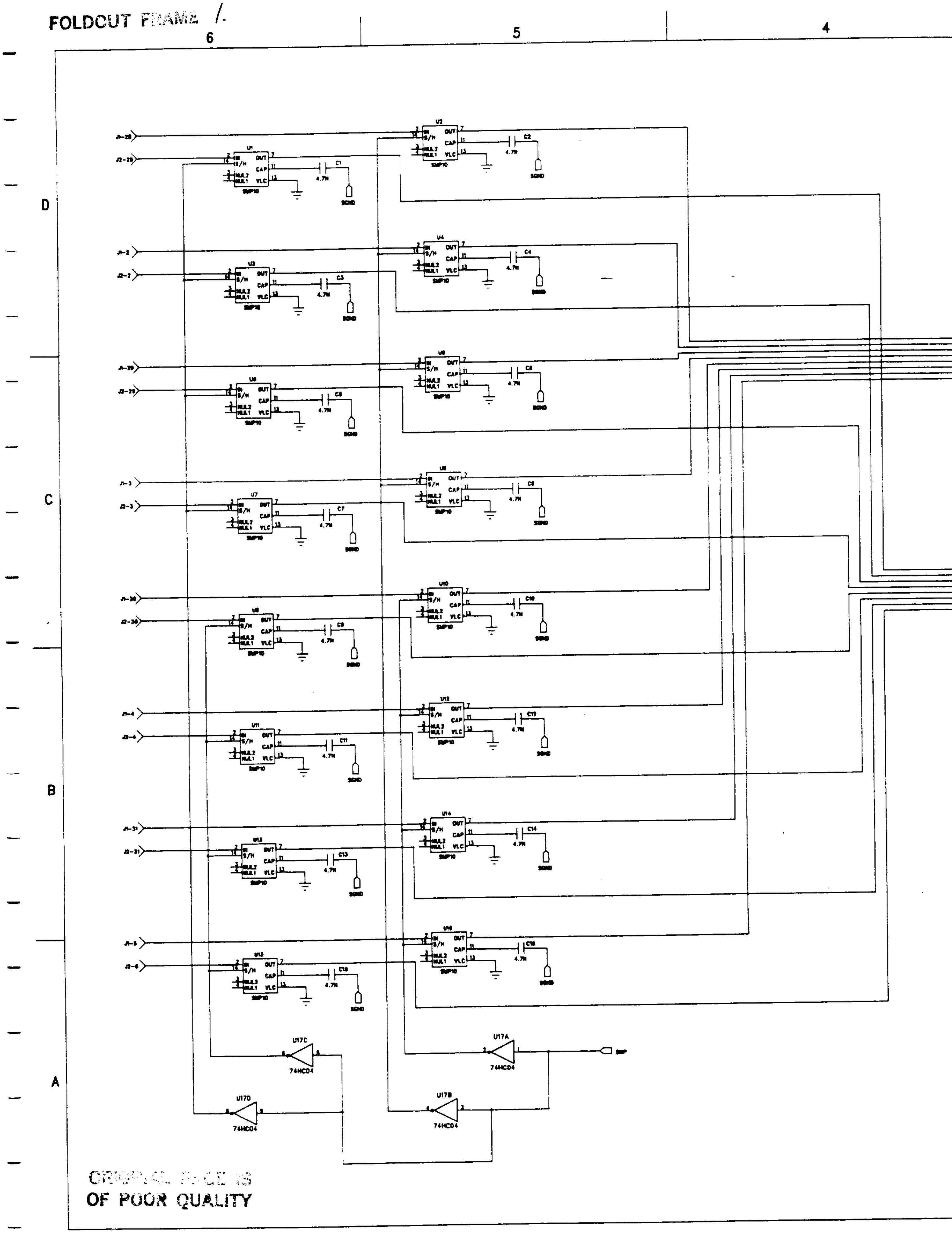
A



ORIGINAL PAGE IS OF POOR QUALITY

		COMPANY: DYNAMICS TECHNOLOGY, INC.			
DRAWN:	DATED:	TITLE: M1000A AMPLIFIER PCB ASSEMBLY			
CHECKED:	DATED:	CODE:	SIZE: D	DRAWING NO: M1000A-1	REV: A
QUALITY CONTROL:	DATED:				
RELEASED:	DATED:	SCALE:		SHEET: 1 OF 1	

FOLDOUT FRAME 1.



ORIGINAL PAGE IS
OF POOR QUALITY

FOLDOUT FRAME 2

3 2 1

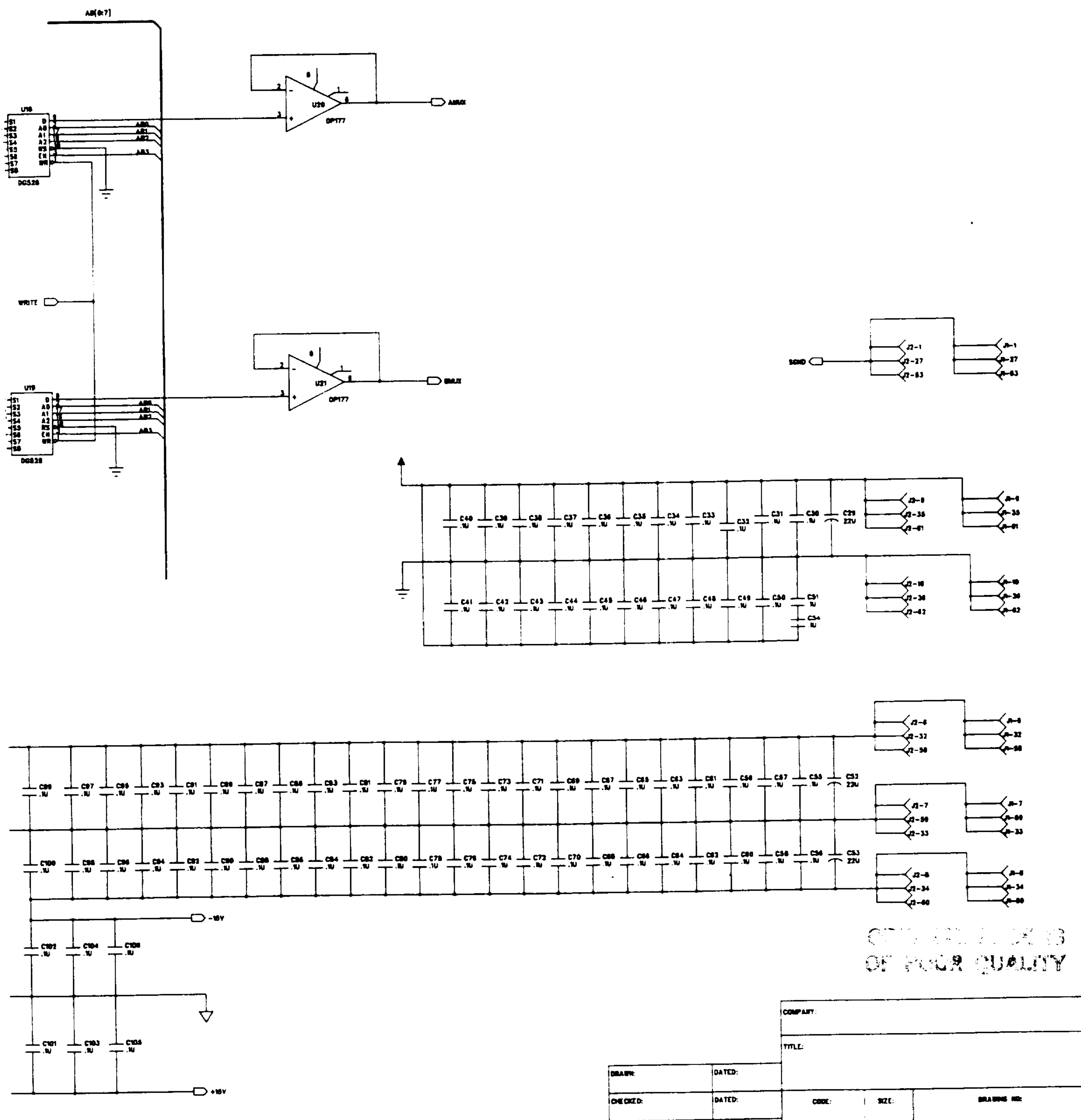
REVISION RECORD			
LTR	ECO NO:	APPROVED:	DATE:

D

C

B

A



ORIGINAL PAGE IS
OF POOR QUALITY

COMPANY:	
TITLE:	
DRAWN:	DATED:
CHECKED:	DATED:
QUALITY CONTROL:	DATED:
RELEASED:	DATED:
CODE:	SIZE:
DRAWING NO:	REV:
SCALE:	SHEET: OF

FOLDOUT FRAME 2

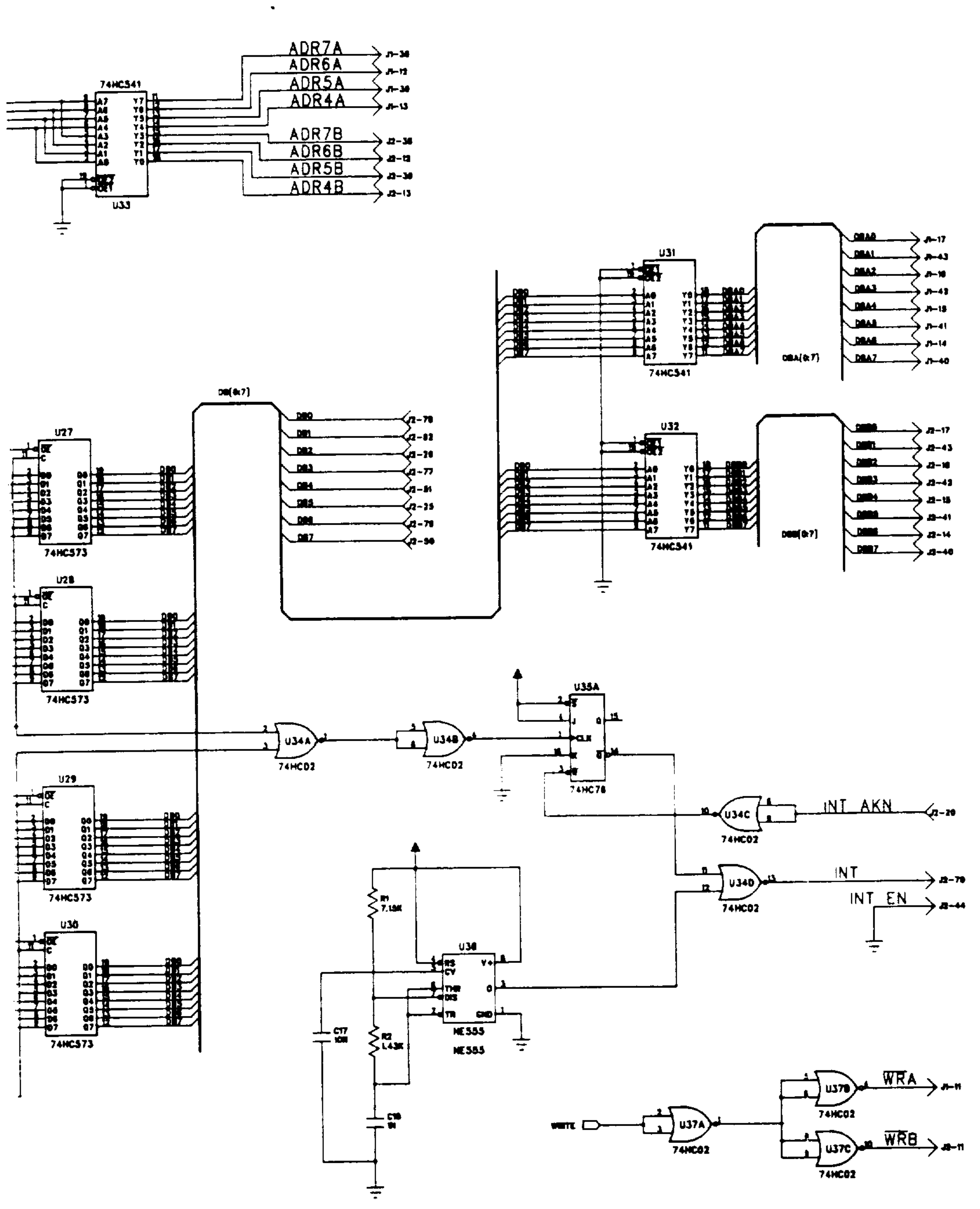
3 2 1

REVISION RECORD			
LTR	ECO NO:	APPROVED:	DATE:

D

C

B

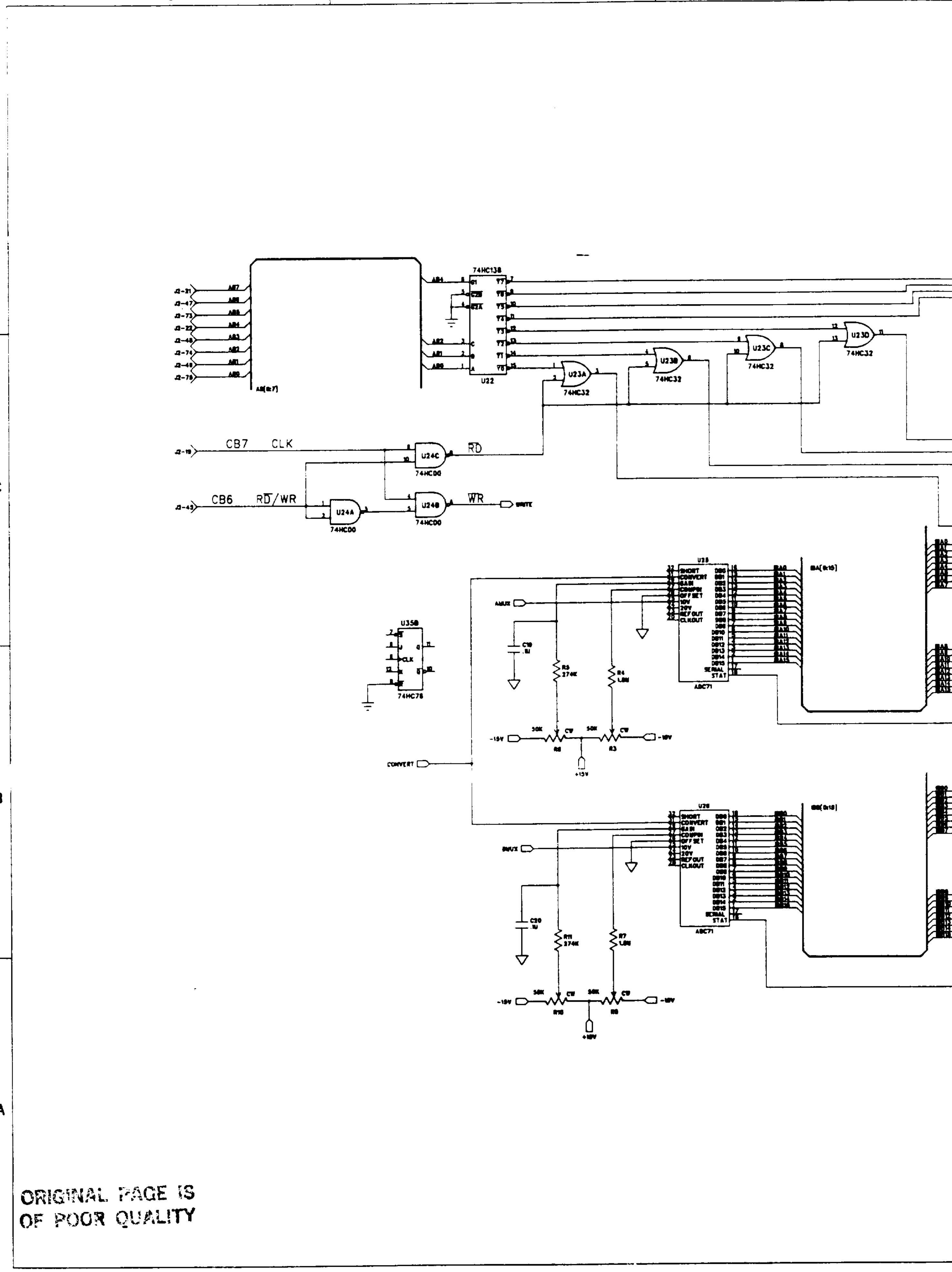


ORIGINAL PAGE IS
OF POOR QUALITY

A

		COMPANY:			
		TITLE:			
DRAWN:	DATED:				
CHECKED:	DATED:	CODE:	SIZE:	DRAWING NO:	REV:
QUALITY CONTROL:	DATED:				
RELEASED:	DATED:	SCALE:		SHEET: OF	

FOLDOUT FRAME 1



ORIGINAL PAGE IS
OF POOR QUALITY

Dynamics Technology, Inc.

DTW-8944-91003

APPENDIX B

Sampled-Line Network Analyzer Installation and Operating Manual

Table of Contents

1. Getting Started

Hardware Installation

The sampled-line network analyzer consists of a microwave sampling module, a post-detection electronics unit (PDEU), and a microwave signal generator. It is designed to work with an IBM PC-AT or clone. Installation is accomplished by interconnecting the above systems and interfacing them to the PC.

Interface between the PC and the network analyzer system is through two plug-in cards in the PC backplane. One board is a Data Translation DT-2801 series board. Any member of the series is acceptable: the interface uses only the 16-bit parallel digital I/O feature, which is common to all boards in the series. The other interface board is a Metrabyte PIO-12 24-bit parallel interface. The DT-2801 controls the system signal generator, while the PIO-12 interfaces with the post-detection electronics.

To install the network analyzer, first install a DT-2801 series board into the PC. The settings of jumpers W3-W16 and W25-W30 will not affect the operation of the signal generator. These jumpers are concerned with the A/D and D/A conversion functions and DMA channel selection. The signal generator uses none of these functions. The jumpers which *should* be checked are W1, W2, and W17-W24. These jumpers set the port addresses of the DT-2801 registers. The signal generator software expects to find the DT-2801 data and command registers at hex addresses 2EC and 2ED respectively. These are the factory default settings.

Next, install the PIO-12 board. Its base address should be set to hex 200 and all interrupts should be disabled. If, due to hardware conflicts, the port address above cannot be used, the definitions contained in **monitor.h, calib.h**,and **measure.h**must be modified to reflect the port addresses actually used, and the programs must be recompiled.

Once the interface boards are installed, the following cabling must be completed. One ribbon cable must be connected between the signal generator and the DT-2801 board. Another connects the PIO-12 board with the post-detection electronics unit (PDEU). These two cables have different connector types to prevent confusion.

To provide synchronization, a BNC cable must be connected between the PDEU's sync output and the RF blanking input of the signal generator. When looking at the signal generator from the back, the RF blanking input is the rightmost BNC connector. The sampled-line module must now be cabled to the system. Included with the system are two cable bundles which each have a multi-coax connector on one end and five BNC males on the other. Plug one multi-coax connector into the sampled-line module and the other into the *top* multi-coax connector on the back of the PDEU. Then connect the BNCs to the bias box. NOTE: as the cables are currently labeled, there is a reversal on the two sides of the bias box. Connect cables on one side of the box in a 1-2-3-4-5 sequence, and those on the other side in a 5-4-3-2-1 sequence.

The only remaining cable is the rf cable between the signal generator and the sampled-line module. In connecting this cable be sure to include a coaxial shield DC block. This interrupts low-frequency ground return currents which can cause large offsets in the sampled-line's detector outputs. In general, attenuators will also be needed between the signal generator and the sampling module to achieve a suitable microwave signal level.

Software Installation

The executable files **monitor.exe, calib.exe,** and **measure.exe** constitute all the software needed to run the network analyzer. Copy them to the directory of choice, and software installation is complete.

Operation

The network analyzer is operated by running the three programs, **monitor.exe, calib.exe,** and **measure.exe,** in sequence. The program **monitor** is used to set bias and power levels and check overall system integrity. The program calib.exe performs a system calibration and saves the results to a file. The program **measure.exe** reads the calibration file, makes a measurement and stores the result in a file.

To put the system in monitor mode, turn on the signal generator, PDEU, and computer, change to the appropriate directory, and type "**monitor.**" The system should now be up and running. The screen should clear and a block of text similar to that shown in Figure 1 should appear left justified and centered top-to-bottom on the screen.

```
          Frequency:      1.000 GHz
          Freq. Inc.:     0.100 GHz
          Step:           U/D

          Bias Curr.:     0.4 µA
          Int. Time:      10 ms
          Range:          1 mV

    X.XXXX    X.XXXX    X.XXXX    X.XXXX    X.XXXX
```

Figure 1. When **monitor** is operating, you should see the above on the PC display. The numbers denoted by the X.XXXX's are the values of the outputs of each of the detectors in the sampled-line module. They are updated in real time.

The system is operated by stepping around the various fields of these displays, and changing them from the keyboard. Changes made on the screen are continuously written out to the hardware.

The desired frequency and frequency step size are set by moving the cursor to the appropriate numeric field with the up and down arrow keys and entering the desired values. The program will accept arrow keys, numbers, the decimal point, INS, DEL, and BACKSPACE, so the displayed value may be edited to whatever value is desired. Whatever changes are made on the display will immediately be written out to the hardware, provided the quantities displayed are within the capabilities of the signal generator.

There is another feature of the frequency control function which is not evident from the display. It is "filter tweaking." For maximum output, the center frequency of the signal generator's tracking filter must be precisely aligned with the oscillator output frequency. This alignment is normally performed by interpolating between values in a look-up table which is maintained in a file accompanying the control program. The table was derived by aligning the system at evenly spaced points in the signal generator's frequency range, and noting the signal generator register values at these points.

The filter adjustment function allows the output power of the signal generator at a given frequency to be peaked by finely adjusting the filter's output passband to center on the oscillator output frequency. The **F1** key moves the filter passband down in frequency. Each key press moves the filter by one **LSB** of its control word.

The other fields on the control screen provide control of the PDEU's gain, integration time, and bias supply. These fields are modified by positioning the cursor to the desired field and using **PgUp** and **PgDn** to step through the allowed values. Actually, the bias current function has been replaced by the external bias box which allows continuous adjustment of the bias current instead of the two values originally designed in. Thus, changing the bias current on the screen has no effect on the actual bias current applied.

The last elements in the display of the **monitor** screen are the numbers across the bottom. These are the actual voltage outputs, in microvolts, of the five detectors in the network analyzer. They are read and updated approximately once each integration time. The user may notice these numbers undergoing a large transient whenever system gain or integration time are changed. This is a normal response to the change in system configuration, and the values should return to stable levels within an integration time or so.

An additional feature of the **monitor** program is the ability to write the detector outputs displayed on the screen to a file. At start-up, the **monitor** program opens an ASCII file called **values.dat.** Anytime during the program's operation, the user may press the "w" key and the current frequency and the five output values will be written to the file.

With the monitor program running, the user should verify that all detectors in the system are responding, and that their outputs are not saturating the PDEU. A good test is to place a sliding short on the device's test port, slide it repeatedly from end to end, and verify that all detector outputs move in response. The PDEU range and integration time, and the diode bias, should then be adjusted so that the detector outputs over the frequency range of interest use most of the output range but never saturate the PDEU. If adjustments of the above three quantities do not yield an output in range, the attenuators in the microwave line feeding the sampled-line module must be adjusted.

Note the proper range once it has been found. The program **calib** will prompt for it.

To exit from the **monitor**program at any point, use the **"ESCAPE"** key.

To start the calibration procedure, execute **calib.exe.** This program will prompt for PDEU gain and frequency range. It will use the maximum integration time, 1 second, to maximize accuracy during calibration. The program **calib.exe**steps through the calibration procedure, prompting the user at each step for whatever calibration standard is required. On ending, it writes the raw calibration data to a file specified by the user.

The program **measure.exe**reads the calibration file and, using the frequency range, settings, and calibration data therein, measures the reflection coefficient of the device under test over one frequency sweep. The reflection coefficient data is stored to an ASCII file specified by the user. An arbitrary number of measurements can be made using the same calibration file. All the algorithms for calibration and measurement are contained in **measure.exe.** The program **calib.exe**just takes data and stores it in a file. Thus, if the user wishes to modify models of the standards or calibration algorithms, he should edit **measure.c.**

2. Hardware

The hardware of the sampled-line network analyzer is described in detail in the text of the final report. Here we present an overview of the digital interface between the PC, the signal generator, and the PDEU. Understanding of these interfaces will aid the user who wishes to make any changes to the control programs above.

Interface Logic

Two interfaces are used in the operation of the network analyzer. The first is that between the PC and the signal generator, and the second is that between the PC and the PDEU.

The YIG-tuned oscillator, voltage-controlled oscillator and YIG-tuned filter in the signal generator are each equipped with digital drivers which each accept a 12-bit input value. Information on the positions of the SPDT microwave relay requires another bit. The control information for all these devices is stored in six latched internal registers in the signal generator.

The functions of these six internal registers of the signal generator are given below.

Register 0:	b7..b0	YTF least significant byte.
Register 1:	b7..b0	YTO least significant byte.
Register 2:	b3..b0	YTF upper nibble.
	b7..b4	YTO upper nibble.
Register 3:	b1	VCO/YTO switch. "1" selects YTO; "0" selects VCO.
Register 4:	b7..b0	VCO kease significant byte.
Register 5:	b7..b4	VCO upper nibble.

Accessing the above registers from the two 8-bit ports of the DT-2801 series board is achieved as follows. DT-2801 digital ports 0 and 1 are set to output mode. Both ports drive active low lines, and hence one's complement must be performed prior to writing the ports. All references here are made to the active true signals, with this inversion being kept in mind.

DT-2801 port 0 is the data port. Data are set up 100 μs before clocking and held 100 μs after clocking. DT-2801 port 1 controls the interface unit port select, or demultiplexing, as well as the clocking. The bits of port 1 are broken up as shown below.

b2..b0	Register select.

b3	Clock. Clocking is pulsed. The clock should be set up low for 100 μs, held high for100μs, and then returned low for 100μs. Thus, the clock is normally inactive.

Interface to the PDEU is through a Metrabyte PIO-12 plug-in board. The PIO-12 is a general-purpose 24-bit parallel interface. Programming of the PIO-12 is straightforward: three consecutive I/O port addresses map to the 3 bytes of the parallel interface. These ports are denoted **pa, pb,** and **pc.** The interface is has several internal registers must be individually written and read. In interfacing to the PDEU, port **pa** of the PIO-12 is used as a bidirectional data port, **pb** is used for address, and **pc** is used or clocks and other control functions. Definitions of the PDEU's internal registers are given below.

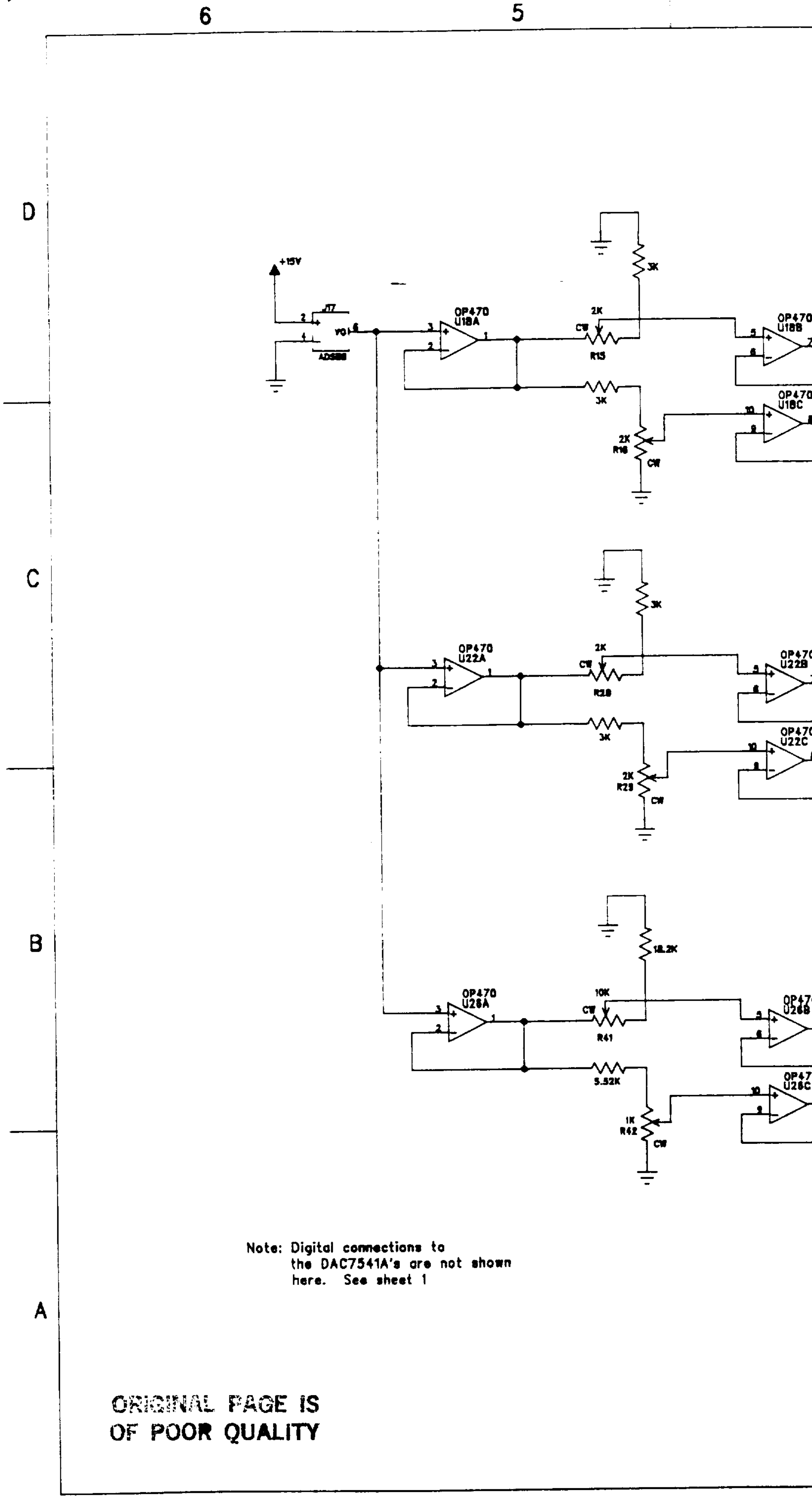
Registers 8-15:	b7..b0	A/D MUX select. A write to register *n* of any value switches the MUX to channel *n* - 8.
Register 20	b7	Bias current.
	b6..b3	DC gain setting.
Register 21	b2-b0	Integration time.
Register 24	b7..b0	A/D converter output, least significant byte.
Register 25	b7..b4	A/D converter output, most significant byte.\hfill}

In setting either gain value or the integration, the binary code written to the PDEU is applied to a set of switches which select a feedback component for the amplifier. Though any value may be written to these switches, the component values in most of the circuits are such that only one of the selected components will dominate in providing feedback for the amplifier. The software provided thus uses only a 1-of-n type selection code. For AC gain, for example, codes of **001, 010,** or **100** are written, with **100** resulting in the highest gain. For bias current a **1** represents high bias, and a **0** represents low.

The bits of the control register, **pc,** are defined below.

b7	Clock. Used to clock data into PDEU for gain and integration settings. Pulsed.
b6	Active low latch line for analog MUX.
b5	Start convert line for A/D converter.
b3	A/D enable.

FOLDOUT FRAME 1.



Note: Digital connections to the DAC7541A's are not shown here. See sheet 1

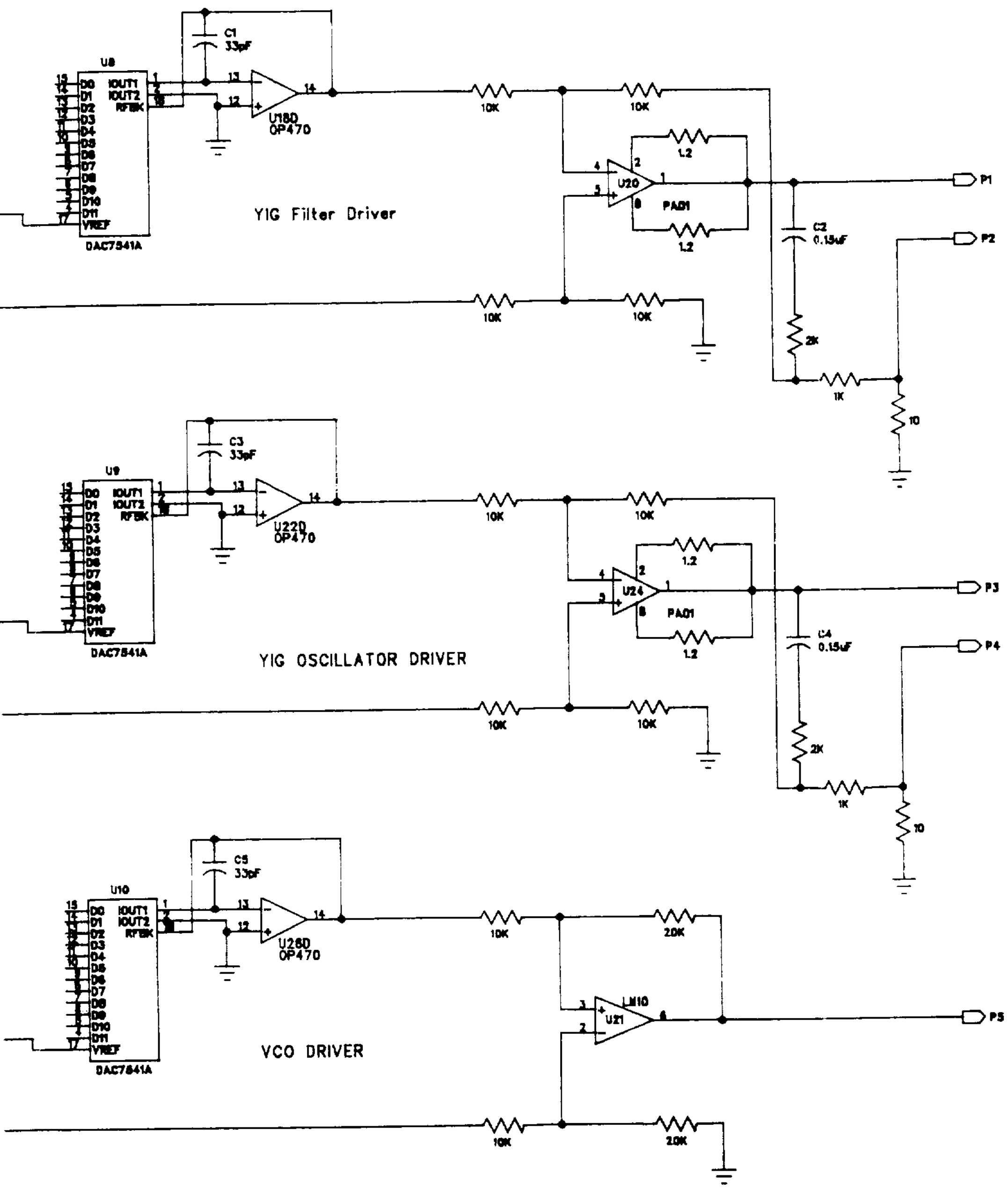
ORIGINAL PAGE IS OF POOR QUALITY

2

	3	2	1

REVISION RECORD

LTR	ECO NO:	APPROVED:	DATE:



D

C

B

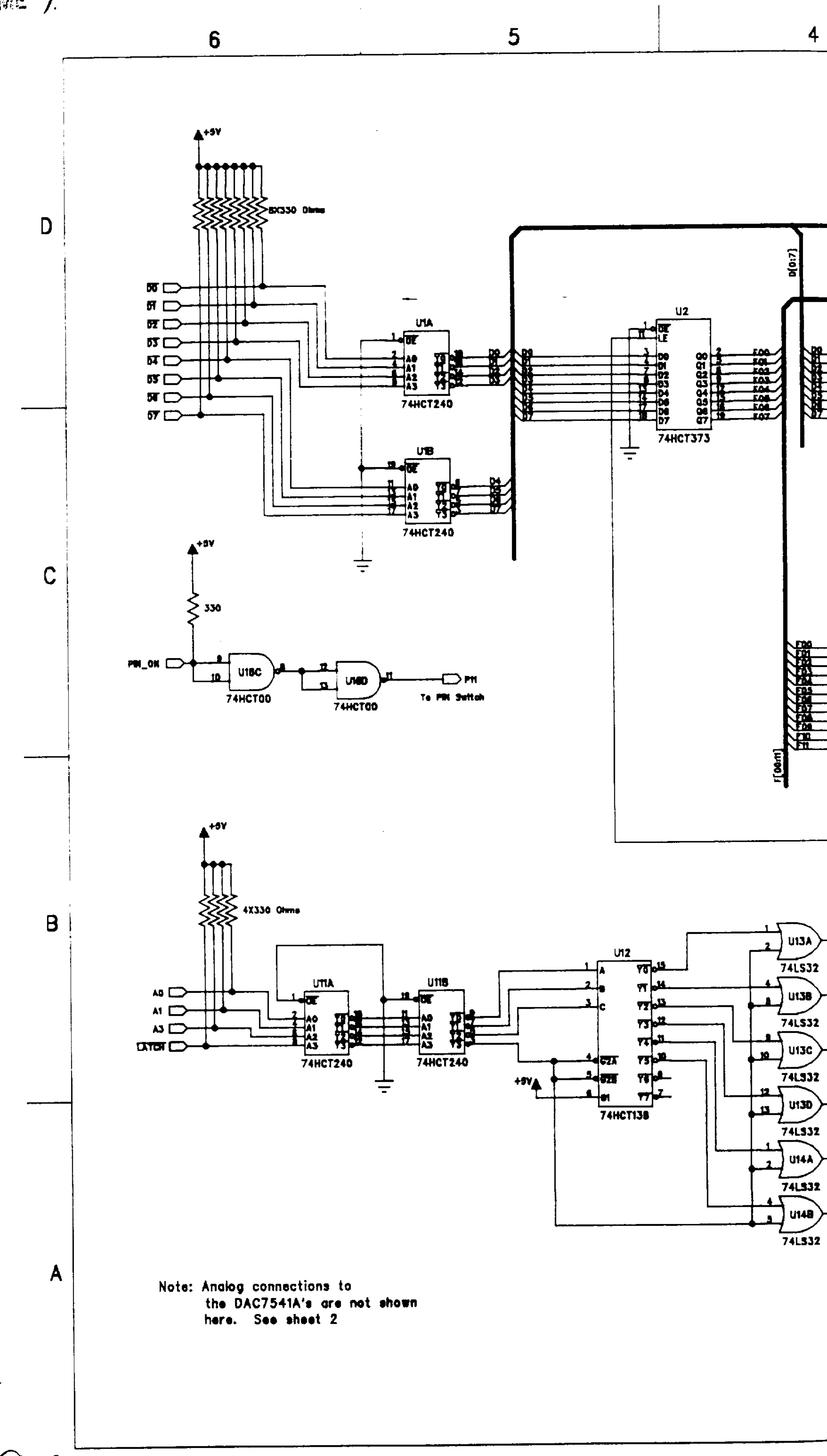
A

ORIGINAL PAGE IS OF POOR QUALITY

COMPANY:	Dynamics Technology, Inc.			
TITLE:	Microwave Signal Generator Driver Circuits			
CODE:	SIZE: C	DRAWING NO:		REV:
SCALE:			SHEET: 2 OF 2	

DRAWN: C.R. Ragsdale	DATED: 1/28/91
CHECKED:	DATED:
QUALITY CONTROL:	DATED:
RELEASED:	DATED:

FOLDOUT FRAME /



Note: Analog connections to
the DAC7541A's are not shown
here. See sheet 2

ORIGINAL PAGE IS
OF POOR QUALITY

C-2

FOLDOUT FRAME 2

3 2 1

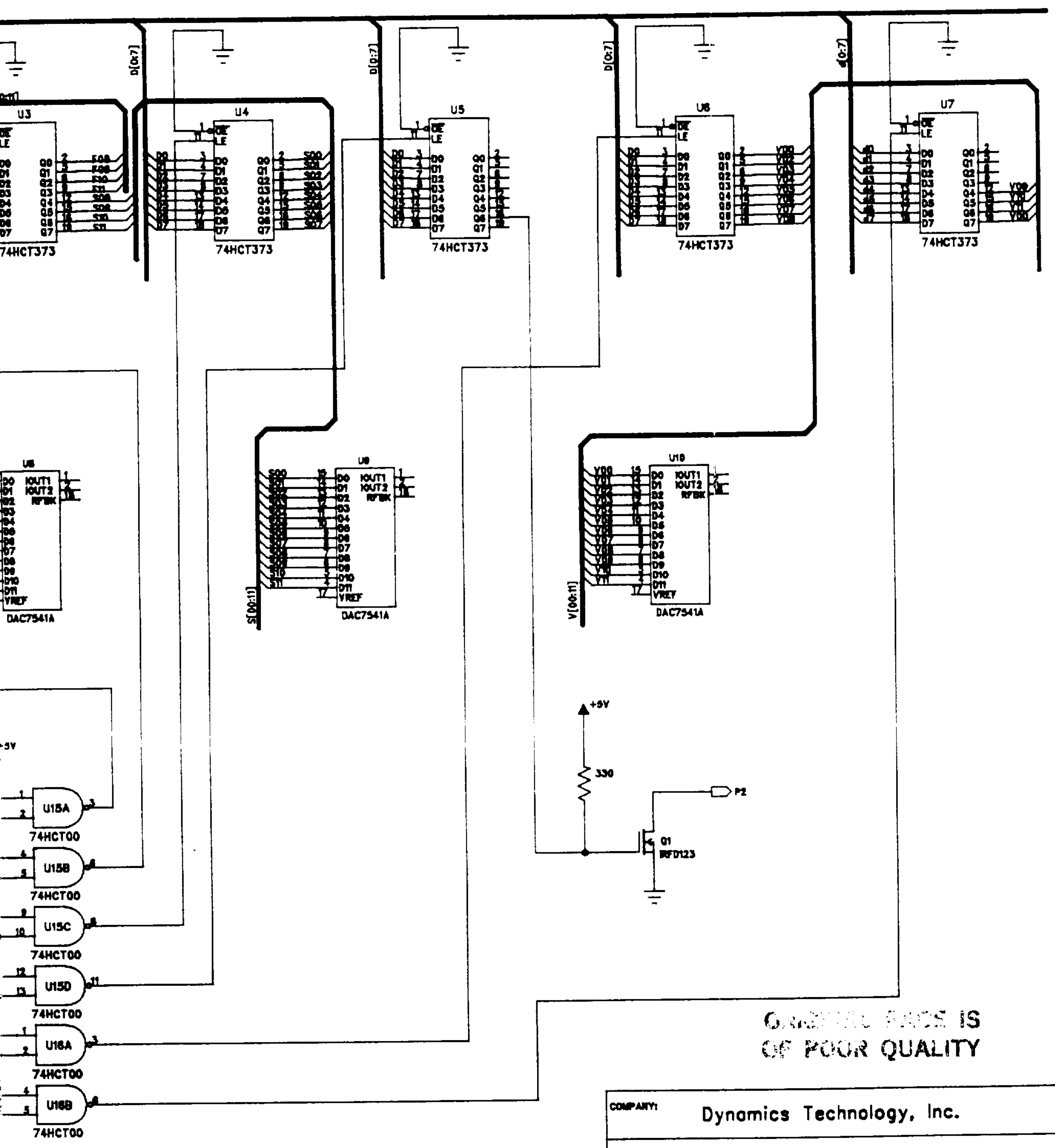
REVISION RECORD			
LTR	ECO NO:	APPROVED:	DATE:

D

C

B

A

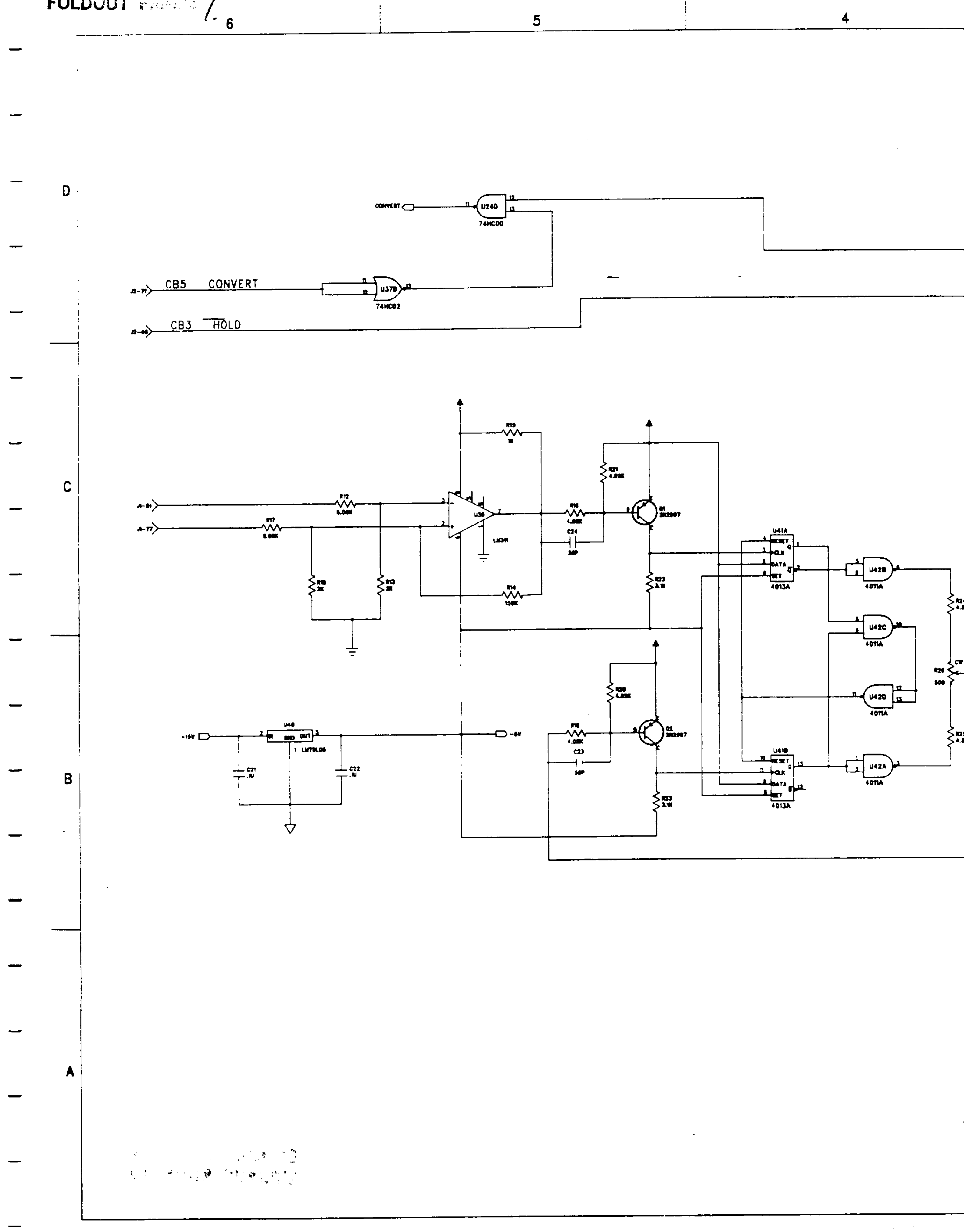


ORIGINAL PAGE IS
OF POOR QUALITY

COMPANY: Dynamics Technology, Inc.			
TITLE: Microwave Signal Generator Digital Interface			
CODE:	SIZE: C	DRAWING NO:	REV:
SCALE:		SHEET: 1 OF 2	

DRAWN: C.R. Ragsdale	DATED: 1/28/91
CHECKED:	DATED:
QUALITY CONTROL:	DATED:
RELEASED:	DATED:

FOLDOUT FRAME 1



3 2 1

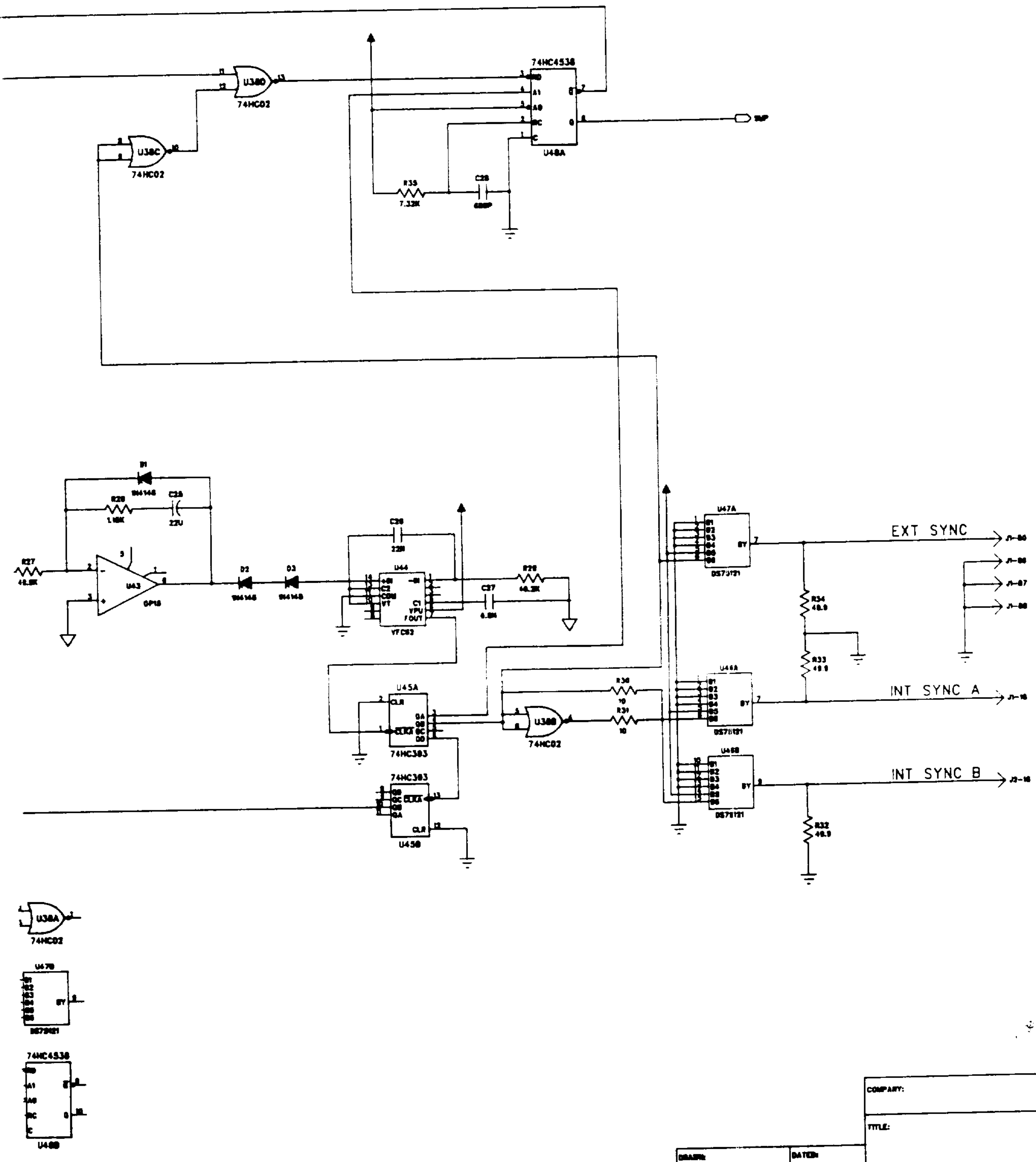
REVISION RECORD			
LTR	ECO NO:	APPROVED:	DATE:

D

C

B

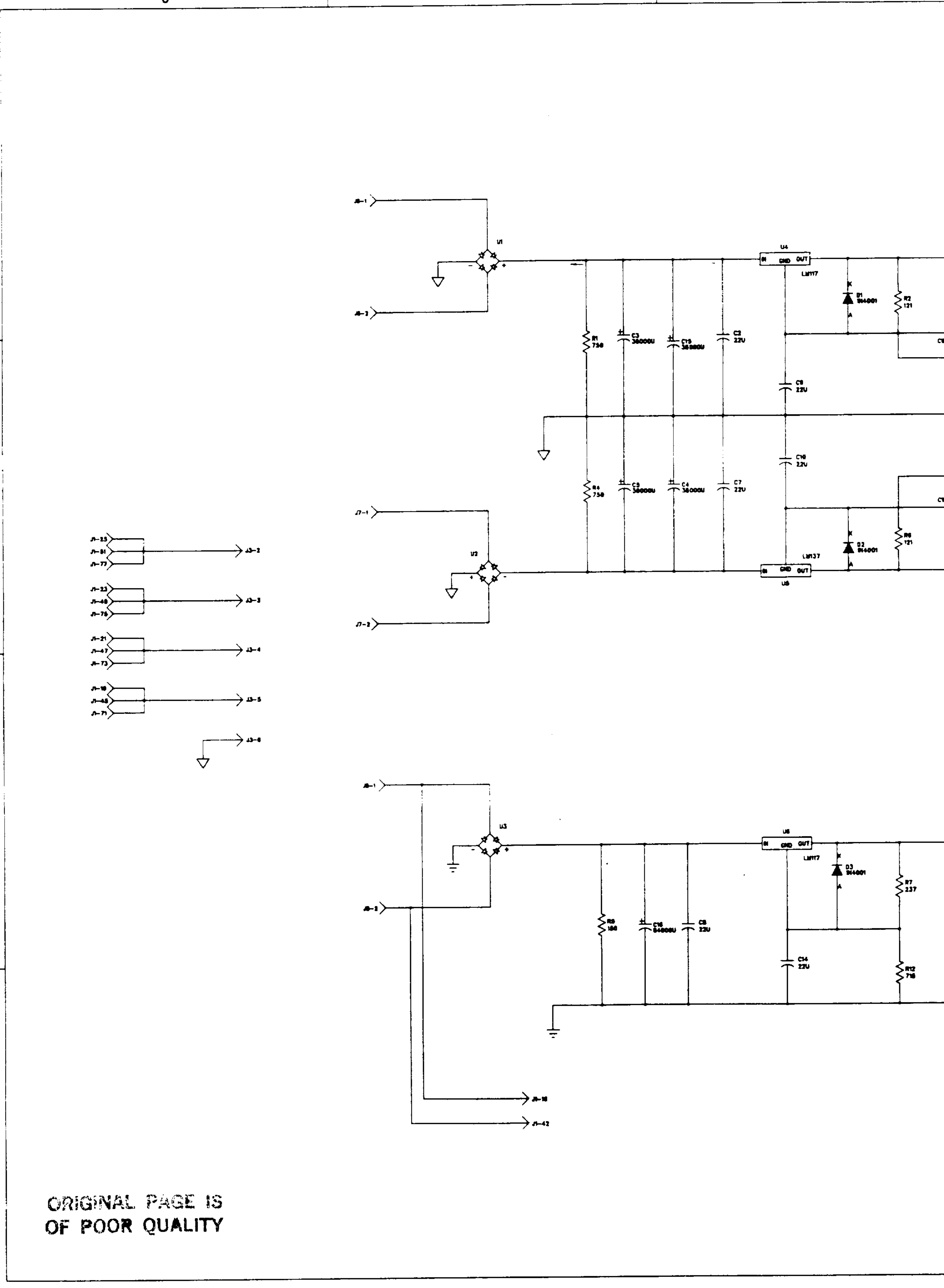
A



COMPANY:	
TITLE:	

DRAWN:	DATED:				
CHECKED:	DATED:	CODE:	SIZE:	DRAWING NO:	REV:
QUALITY CONTROL:	DATED:				
RELEASED:	DATED:	SCALE:		SHEET: OF	

FOLDOUT FRAME /.



ORIGINAL PAGE IS
OF POOR QUALITY

2

3 2 1

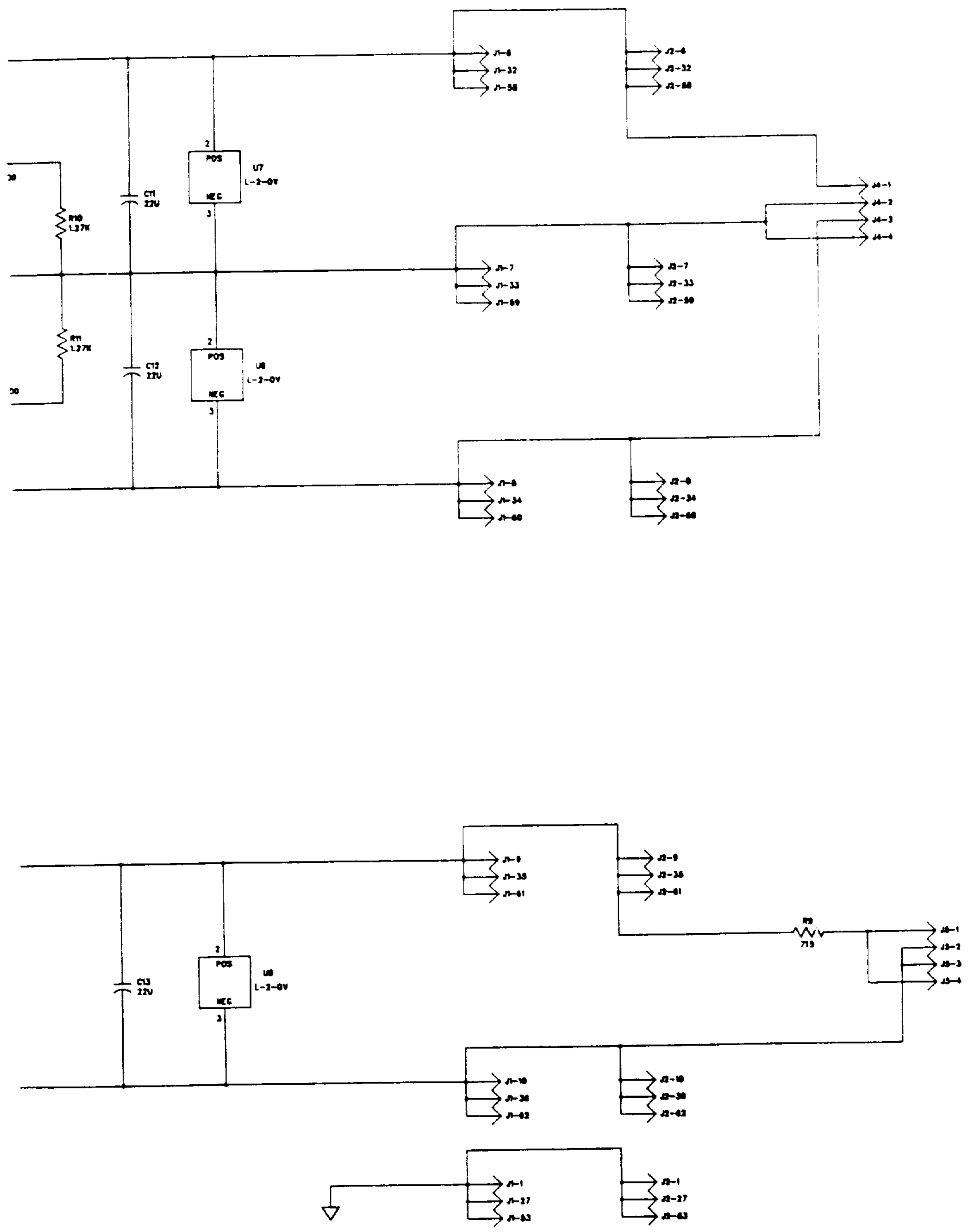
REVISION RECORD			
LTR	ECO NO:	APPROVED:	DATE:

D

C

B

A



ORIGINAL PAGE IS
OF POOR QUALITY

		COMPANY: DYNAMICS TECHNOLOGY, INC.			
		TITLE: M1000A POWER SUPPLY PCB ASSEMBLY			
DRAWN:	DATED:				
CHECKED:	DATED:	CODE:	SIZE: D	DRAWING NO: M1000A-3	REV: A
QUALITY CONTROL:	DATED:				
RELEASED:	DATED:	SCALE:		SHEET: 1 OF 1	

3. Software

Program Description

A:l the control programs are is written in Turbo C. The programs calib.c} are structurally simple, both executing a single list of instructions then exiting. Program **monitor**consists of a central loop which is continuously repeated. A "pseudo-language" representation of the main loop is given below.

```
loop{
    if keyboard is hit{
        if input is ESC key then exit
        if input is a valid character, invoke line editor and update screen
        if input is cursor key, move cursor
        read data from computer screen and update internal data
        }
    write updated data to signal generator
    if PDEU data has changed, write to PDEU
}
```

If no keyboard activity is detected, the program just continuously updates the registers of the signal generator interface. As you type inputs, a line editor modifies the data on the screen, then the program reads these data off the screen directly, so what you see is what you get. A complete listing of all programs is presented on the following pages.

```
#include "monitor.h"

/*********************************************************************/
/*                                                                   */
main()
/*                                                                   */
/* This program gives a real-time display of the outputs of the      */
/* network analyzer's detectors and allows the user to adjust the    */
/* frequency of operation, system gain, and diode bias current to    */
/* achieve optimum performance.                                      */
/*                                                                   */
/*********************************************************************/

{

    int curx = 19, cury = 12, exit = 0, j;
    union REGS a, b;
    char in_char, ascii1, ascii2, insert_mode = 0, change_settings = 0;
    struct inst_status status = {1, 1, 1, 1.0, 0.1};
    char g1[8] = {0, 1, 2, 3, 2, 3, 2, 3};
    char g2[8] = {0, 1, 1, 1, 2, 2, 3, 3};
    FILE *streamer, *streamer1;
    unsigned int out_vector[5];
    char *int_time[4] = {" ", "10 ms  ", "100 ms ", "1 s   "};
    char *range[8] = {" ", "1 mV  ", "300 uV", "100 uV", "30 uV ", "10 uV  ",
    "3 uV   ", "1 uV   "};
    char *bias_curs[2] = {"0.4 uA ", "10 uA "};
    double gain_vals[8] = {1.0, 10.4933, 0.033179, 0.10493, 0.331790,
                          1.04933, 3.31790, 10.4933}, voltage;
    char *formats[11] = {" ", "%6.4lf", "%5.1lf", "%5.1lf", "%5.2lf",
                         "%5.2lf", "%5.3lf", "%6.4lf"};

    calstore = (int*) malloc(4096*sizeof(int));
    clrscr();
    dt_2801_set_up();

    /* Read in signal generator alignment calibration file. */

    streamer = fopen("sweepcal.dat","rt");
    for (j = 0; j < 4096; ++j)
          fscanf(streamer, " %X", &(calstore[j]));
    fclose(streamer);

    /* Read settings from last run of program and restore program to */
    /* previous state.                                                */
    streamer = fopen("settings.dat","rt");
    streamer1 = fopen("values.dat","wt");
    if (streamer){
          fscanf(streamer," %d %d %d %lf %lf", &(status.range),
          &(status.integ), &(status.bias_current),
          &(status.frequency), &(status.freq_step));
          fclose(streamer);
    }

    /* Set up program per previous settings or defaults */

    initialize_screen(status, curx, cury);
    to_frequency(status.frequency);
    data_out();
    set_gains_int(g1[status.range],g2[status.range], status.integ,
                    status.bias_current);
    data_in();

    /* Begin main keyboard scan loop */

    while (exit == 0){
        if (kbhit() != 0){

          /* if user hits ESC, write status and exit program */

            if ((in_char = getch()) == 0x1b){
               exit = 1;
               streamer = fopen("settings.dat","wt");
               fprintf(streamer,"%d\n%d\n%d\n%lf\n%lf\n",
                    status.range, status.integ, status.bias_current,
                         status.frequency, status.freq_step);
               fclose(streamer);
          }

        /* if user hits a numeric key, and cursor is positioned */
          /* to a numeric field, edit field                       */

            else if (((in_char >= '0') && (in_char <= '9'))
                 || (in_char == '.') || (in_char == 0x08)
                     || (in_char == ' ')){
                 if ((cury == 10) || (cury == 11))
                     edit_line(&curx, &ccury, in_char, insert_mode);
            }
```

```
/* if user inputs 'w' write current diode readings to file */

else if (in_char == 'w'){
 fprintf(streamer1, "%lf ", status.frequency);
 for (j = 0; j < 5; ++j){
      voltage = ((double)out_vector[j])/65535.0*10.0
                    / gain_vals[status.range];
      fprintf(streamer1, formats[status.range], voltage);
      fprintf(streamer1, "    ");
 }
 fprintf(streamer1,"\n");
}

  else if (in_char == 0){ /* 0 is prefix for arrow, function keys */
      in_char = getch();
      switch(in_char){

     /* on left arrow move cursor left (within limits) */

          case 0x4b : if (((cury == 10) || (cury == 11))
                 && (curx > 19)) --curx;
                    break;

     /* on right arrow, move cursor right */

          case 0x4d : if (((cury == 10) || (cury == 11))
                 && (curx < 24)) ++curx;
                    break;

     /* on up arrow, move cursor up */

          case 0x48 : if (cury == 10){
                          curx = 19;
              cury = 16;
              }
          else if (cury == 14) cury = 12;
                    else --cury;
                    break;

     /* on down arrow move cursor down */

          case 0x50 : if (cury == 11){
                           curx = 19;
```

```
            cury = 12;
        }
        else if (cury == 12) cury = 14;
        else if (cury == 16) cury = 10;
        else ++cury;
                break;

/* Insert key toggles insert mode */

     case 0x52 : insert_mode ^= 0x01;

      /* change cursor style to fit insert mode */

                if (insert_mode == 0){
                    a.h.ah = 0x01;
                    a.h.ch = 0x06;
                    a.h.cl = 0x07;
                    int86(0x10,&a,&b);
                }
                if (insert_mode == 1){
                    a.h.ah = 0x01;
                    a.h.ch = 0x00;
                    a.h.cl = 0x06;
                    int86(0x10,&a,&b);
                }
                break;

/* on delete, call line editor to modify screen */

     case 0x53 : if ((cury == 10) || (cury == 11))
                    edit_line(&curx, &cury,
             0x7F, insert_mode);
                break;

/* on PgUp, increase sweeper frequency, bias current, */
/* gain or integration time, depending on cursor      */
/* position.                                          */

     case 0x49 : if (cury == 12){ /* increase frequency */
          status.frequency += status.freq_step;
          if (status.frequency > 8.0)
              status.frequency = 8.0;
          if (status.frequency < 1.0)
              status.frequency = 1.0;
            /* make cursor invisible to update */
            /* frequency value on screen       */

                        a.h.ah = 0x01;
                        a.h.ch = 0x20;
                        a.h.cl = 0x00;
                        int86(0x10,&a,&b);
            gotoxy(19,10);
            printf("%6.3lf",status.frequency);
            gotoxy(curx,cury);

            /* restore cursor with correct style */

                        if (insert_mode == 0){
                            a.h.ah = 0x01;
                            a.h.ch = 0x06;
                            a.h.cl = 0x07;
                            int86(0x10,&a,&b);
                        }
                        else{
                            a.h.ah = 0x01;
                            a.h.ch = 0x00;
                            a.h.cl = 0x06;
                            int86(0x10,&a,&b);
            }
                   }
        else if (cury == 14){/*increase bias current*/
                        ++status.bias_current;
            if (status.bias_current > 1)
                 status.bias_current = 0;
            printf("%s",
                 bias_curs[status.bias_current]);
                   }
        else if (cury == 15){ /* increase int. time */
           ++status.integ;
           if (status.integ > 3)
                status.integ = 1;
            printf("%s",
                 int_time[status.integ]);
            ++change_settings;
           }
        else if (cury == 16){ /* increase gain */
           --status.range;
```

```
                if (status.range < 1)
                     status.range = 7;
                 printf("%s",
                      range[status.range]);
                 ++change_settings;
                }
            break;

        /* on PgDn, decrease sweeper frequency, bias current, */
        /* gain or integration time, depending on cursor      */
        /* position.                                          */

            case 0x51 : if (cury == 12){ /* decrease frequency */
                 status.frequency -= status.freq_step;
                 if (status.frequency > 8.0)
                     status.frequency = 8.0;
                 if (status.frequency < 1.0)
                     status.frequency = 1.0;

                 /* cursor invisible */

                             a.h.ah = 0x01;
                             a.h.ch = 0x20;
                             a.h.cl = 0x00;
                             int86(0x10,&a,&b);
                 gotoxy(19,10);
                 printf("%6.3lf",status.frequency);

                 /* restore cursor */

                 gotoxy(curx,cury);
                             if (insert_mode == 0){
                                 a.h.ah = 0x01;
                                 a.h.ch = 0x06;
                                 a.h.cl = 0x07;
                                 int86(0x10,&a,&b);
                             }
                             else{
                                 a.h.ah = 0x01;
                                 a.h.ch = 0x00;
                                 a.h.cl = 0x06;
                                 int86(0x10,&a,&b);
                 }
        }
```

```
                else if (cury == 14){/*decrease bias current*/
                                --status.bias_current;
                    if (status.bias_current < 0)
                         status.bias_current = 1;
                    printf("%s",
                         bias_curs[status.bias_current]);
                            }
                else if (cury == 15){/*decrease int. time*/
                   --status.integ;
                   if (status.integ < 1)
                        status.integ = 3;
                   printf("%s",
                         int_time[status.integ]);
                    ++change_settings;
                   }
                else if (cury == 16){/* decrease gain */
                   ++status.range;
                   if (status.range > 7)
                        status.range = 1;
                   printf("%s",
                         range[status.range]);
                    ++change_settings;
                   }
                break;

          /* The function keys F1, F2, and F3 allow the user   */
          /* to adjust the YTF offset to peak signal generator */
          /* output power.                                     */

          case 0x3b : --YTFTWEAK;     /* F1 */
                      break;
          case 0x3c : ++YTFTWEAK;     /* F2 */
                      break;
          case 0x3d : YTFTWEAK = 0;   /* F3 */
                      break;
            }
       }
       gotoxy(curx,cury);
     update_data(curx, cury, &status, insert_mode);

     /* Write updated data out to hardware. */

     to_frequency(status.frequency);
     if (change_settings != 0){
            data_out();
            set_gains_int(g1[status.range],g2[status.range], status.integ,
                    status.bias_current);
                data_in();
            }
            change_settings = 0;
          }

        /* Wait for appropriate number of ticks of system clock before */
        /* reading PDEU outputs to avoid oversampling.                 */

        if (((status.integ == 1) && (MOD_ON > 0)) ||
             ((status.integ == 2) && (MOD_ON > 2)) ||
                 ((status.integ == 3) && (MOD_ON > 18))){
            read_ads(3,7,out_vector);

            /* invisible cursor */

            a.h.ah = 0x01;
            a.h.ch = 0x20;
            a.h.cl = 0x00;
            int86(0x10,&a,&b);

            /* write new output values, appropriately scaled */

            for (j = 0; j < 5; ++j){
                 gotoxy(4+10*j,18);
                 voltage = ((double)out_vector[j])/65535.0*10.0
                                / gain_vals[status.range];
                 printf(formats[status.range], voltage);
            }

            /* restore cursor */

            gotoxy(curx,cury);
            if (insert_mode == 0){
                 a.h.ah = 0x01;
                 a.h.ch = 0x06;
                 a.h.cl = 0x07;
                 int86(0x10,&a,&b);
            }
            else{
                 a.h.ah = 0x01;
                 a.h.ch = 0x00;
```

```
                  a.h.cl = 0x06;
                  int86(0x10,&a,&b);
              }
              MOD_ON = 0; /* reset timer tick counter */
           }
        }
        fclose(streamer1);
}

/* end of main routine */

/**********************************************************************/
/*                                                                    */
void wait_microsec(int arg1)
/*                                                                    */
/* Assembly language timing loop for short delays.                    */
/*                                                                    */
/**********************************************************************/

{
asm   mov  cx,arg1
s1:
asm   nop
asm   loop s1
}

/**********************************************************************/
/*                                                                    */
void mode_set(char mode)
/*                                                                    */
/* Sets mode of programmable peripheral interface on PIO-12 board.   */
/*                                                                    */
/**********************************************************************/

{
      outportb(pio12_control, mode);
asm   jmp  $+2
}
```

```
/***********************************************************************/
/*                                                                     */
void set_gains_int(char g1, char g2, char int1, char bias)
/*                                                                     */
/* This routine sets the gains of the various programmable ampli-     */
/* fiers in the network analyzer's post-detection electronics.  It    */
/* also sets the integration time and the bias current.               */
/* Communication with the post-detection electronics is through       */
/* the PIO-12 24-bit parallel interface card.                         */
/*                                                                     */
/***********************************************************************/

{
        outportb(pio12_pb, 0x14);
asm     jmp  $+2
        outportb(pio12_pa,
             ~((0x01 << (g1-1)) | (0x08 << (g2-1)) |
                                   ((bias << 7) & 0x80)));
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2
        outportb(pio12_pc, 0x80);
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2

        outportb(pio12_pb, 0x15);
asm     jmp  $+2
        outportb(pio12_pa, ~(0x01 << (int1-1)));
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2
        outportb(pio12_pc, 0x80);
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2
}
```

```
/************************************************************************/
/*                                                                      */
void data_in()
/*                                                                      */
/* Set PIO-12 board up for input on one of its 8-bit ports.  Keep       */
/* other 2 ports in output mode for control.                            */
/*                                                                      */
/************************************************************************/

{
        char mode;

        outportb(pio12_pa, 0x00);
asm     jmp  $+2
        outportb(pio12_pb, 0x00);
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2

        mode = pa_is_input | pb_is_output | pclo_is_output
               | pchi_is_output | pa_pchi_is_mode_0 | pb_pclo_is_mode_0
                     | mode_set_active;
        mode_set(mode);
}

/************************************************************************/
/*                                                                      */
void data_out()
/*                                                                      */
/* Set PIO-12 board up for output on all of its 8-bit ports.            */
/*                                                                      */
/************************************************************************/

{
        char mode;

        mode = pa_is_output | pb_is_output | pclo_is_output
               | pchi_is_output | pa_pchi_is_mode_0 | pb_pclo_is_mode_0
                     | mode_set_active;
        mode_set(mode);
}
```

```
/***********************************************************************/
/*                                                                     */
void read_ads(char s_channel, char e_channel, unsigned int *out_vector)
/*                                                                     */
/* Uses the 16-bit A/D converter in the post-detection electronics    */
/* to read the analog outputs of the various synchronous detectors.   */
/* s_channel is first channel to be read, e_channel is last.  Array   */
/* of converted values is returned in out_vector.                      */
/*                                                                     */
/***********************************************************************/

{
        char byte1[2], i;
        unsigned int   *temp;

        temp = (unsigned int *) byte1;

        outportb(pio12_pc, 0xC0);
        delay(2);
        outportb(pio12_pc, 0xC8);
        wait_microsec(100);
        for (i = s_channel; i <= e_channel; ++i){
             outportb(pio12_pb, 0x08 | (i & 0x07));
asm     jmp  $+2;
             outportb(pio12_pc, 0x88);
asm     jmp  $+2;
             outportb(pio12_pc, 0xC8);
             wait_microsec(10);
             outportb(pio12_pc, 0xE8);
asm     jmp  $+2;
             outportb(pio12_pc, 0xC8);
             wait_microsec(100);
             outportb(pio12_pb, 0x18);
asm     jmp  $+2;
             byte1[0] = inportb(pio12_pa);
asm     jmp  $+2;
             outportb(pio12_pb, 0x19);
asm     jmp  $+2;
             byte1[1] = inportb(pio12_pa);
             out_vector[i-s_channel] = ~(*temp);
        }
        outportb(pio12_pc, 0xC0);
}
```

```
/***********************************************************************/
/*                                                                     */
void dt_error_check()
/*                                                                     */
/* This routine is called whenever an error is reported by the Data   */
/* Translation board.  It queries the board, interprets the resulting */
/* error code, and prints an error message.                            */
/*                                                                     */
/***********************************************************************/

{
        int  i;
        int  ready = 0;
        int  timeout = -32767;
        char a, b, reg;

        for (i = 0; i < 100; ++i){}
        outportb(dt_2801_command_register, READ_ERR);
        while (ready == 0){
             for (i = 0; i < 100; ++i){}
             reg = inportb(dt_2801_status_register);
             if ((reg & DOR) == DOR)
                  ready = 1;
             else{
                  ++timeout;
                  if (timeout == 32767){
                       printf("Device time-out on DT-2801 board,\n");
                       printf("during dt_error_check.\n");
                       ready = 1;
                  }
             }
        }
        a = inportb(dt_2801_data_register);
        ready = 0;
        timeout = -32767;
        while (ready == 0){
             for (i = 0; i < 100; ++i){}
             reg = inportb(dt_2801_status_register);
             if ((reg & DOR) == DOR)
                  ready = 1;
             else{
                  ++timeout;
                  if (timeout == 32767){
                       printf("Device time-out on DT-2801 board,\n");
```

```
                    printf("during dt_error_check.\n");
                    ready = 1;
                }
            }
        }
        b = inportb(dt_2801_data_register);
        if ((a & 0x02) == 0x02)
            printf("Command Overwrite Error\n");
        if ((a & 0x04) == 0x04)
            printf("Clock Set Error\n");
        if ((a & 0x08) == 0x08)
            printf("Digital Port Select Error\n");
        if ((a & 0x10) == 0x10)
            printf("Digital Port Set Error\n");
        if ((a & 0x20) == 0x20)
            printf("DAC Select Error\n");
        if ((a & 0x40) == 0x40)
            printf("DAC Clock Error\n");
        if ((a & 0x80) == 0x80)
            printf("DAC No. Conversions Value Error\n");
        if ((b & 0x01) == 0x01)
            printf("A/D Channel Error\n");
        if ((b & 0x02) == 0x02)
            printf("A/D Gain Error\n");
        if ((b & 0x04) == 0x04)
            printf("A/D Clock Error\n");
        if ((b & 0x08) == 0x08)
            printf("A/D Multiplexer Error\n");
        if ((b & 0x10) == 0x10)
            printf("A/D No. Conversions Value Error\n");
        if ((b & 0x20) == 0x20)
            printf("Data Where Command Expected Error\n");
}
```

```
/***************************************************************************/
/*                                                                         */
void dt_set_wait(unsigned char bytecode)
/*                                                                         */
/* Part of the handshaking with the Data Translation board, this           */
/* routine waits until a particular byte pattern is set in the DT-         */
/* 2801's status register.                                                 */
/*                                                                         */
/***************************************************************************/

{
      char ready, i, idle;
      int timeout;

      ready = 0;
      timeout = -32767;
      while (ready == 0){
           /*idle = 0;
           while (idle <= 10) ++idle;*/
           i = inportb(dt_2801_status_register);
           if ((i & 0x80) == 0x80) dt_error_check();
           if ((i & bytecode) == bytecode)
                ready = 1;
           else
           {
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
                     printf("during dt_set_wait.\n");
                     ready = 1;
                }
           }
      }
}
```

```
/*************************************************************************/
/*                                                                       */
void dt_clear_wait(unsigned char bytecode)
/*                                                                       */
/* Part of the handshaking with the Data Translation board, this         */
/* routine waits until a particular byte pattern is cleared in the       */
/* DT-2801's status register.                                            */
/*                                                                       */
/*************************************************************************/

{
      char ready, i, idle;
      int timeout;

      ready = 0;
      timeout = -32767;
      while (ready == 0){
           /*idle = 0;
           while (idle <= 10) ++idle;*/
           i = inportb(dt_2801_status_register);
           if ((i & 0x80) == 0x80) dt_error_check();
           if ((i & bytecode) == 0)
                ready = 1;
           else
           {
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
                     printf("during dt_clear_wait.\n");
                     ready = 1;
                }
           }
      }
}
```

```
/**********************************************************************/
/*                                                                    */
void write_digital(unsigned char byte_select, unsigned char out_byte)
/*                                                                    */
/* This I/O routine writes the byte out_byte to the DT-2801 parallel  */
/* port selected by byte_select.  The value of byte_select can be 1   */
/* or 0 since the DT-2801 has only 2 ports.                           */
/*                                                                    */
/**********************************************************************/

{
      int  i;

      outportb(dt_2801_command_register, WRITE_DIG_IMM);
      dt_clear_wait(DIF);
      outportb(dt_2801_data_register, byte_select);
      dt_clear_wait(DIF);
      outportb(dt_2801_data_register, out_byte);
      dt_clear_wait(DIF);
      dt_set_wait(READY);
}
```

```
/***********************************************************************/
/*                                                                     */
void dt_2801_init()
/*                                                                     */
/* This routine initializes the Data Translation board.  It first      */
/* resets the board, then writes values into several control          */
/* registers.                                                          */
/*                                                                     */
/***********************************************************************/

{
      char dummy;

      outportb(dt_2801_command_register,STOP);
      dummy = 0;
      while (dummy <= 100) ++dummy;
      inportb(dt_2801_data_register);
      dummy = 0;
      while (dummy <= 100) ++dummy;
      outportb(dt_2801_command_register,RESET);
        dt_set_wait(DOR);
      inportb(dt_2801_data_register);
      dt_set_wait(READY);
      outportb(dt_2801_command_register,CLEAR_ERROR);
      dt_clear_wait(DIF);
        dt_set_wait(READY);
      outportb(dt_2801_command_register,SET_INT_CLK);
      dt_clear_wait(DIF);
      outportb(dt_2801_data_register,0x1E);/*clock period = 75 us*/
      dt_clear_wait(DIF);
      outportb(dt_2801_data_register,0x00);
      dt_clear_wait(DIF);
      dt_set_wait(READY);
      outportb(dt_2801_command_register,SET_A_D_PARAMS);
      dt_clear_wait(DIF);
      outportb(dt_2801_data_register,0x00);/*gain code*/
        dt_clear_wait(DIF);
        outportb(dt_2801_data_register, 0x00);/*start channel*/
        dt_clear_wait(DIF);
        outportb(dt_2801_data_register, 0x04);/*end channel*/
        dt_clear_wait(DIF);
        outportb(dt_2801_data_register,0x05);/*lo(number of samples)*/
        dt_clear_wait(DIF);
        outportb(dt_2801_data_register,0x00);/*hi(number of samples)*/
```

```
            dt_clear_wait(DIF);
            dt_set_wait(READY);
        outportb(dt_2801_command_register,SET_DIGITAL_OUTPUT);
        dt_clear_wait(DIF);
        outportb(dt_2801_data_register,BOTH);/*set both ports for output*/
        dt_clear_wait(DIF);
        dt_set_wait(READY);
}

/***********************************************************************/
/*                                                                     */
void write_to_YTF(unsigned int value)
/*                                                                     */
/* This I/O routine writes the integer value to the YIG-tuned          */
/* filter's control port.                                              */
/*                                                                     */
/***********************************************************************/

{
        unsigned char a;

        value += YTFTWEAK;
        YTFLSB = value;
        write_digital(0, ~YTFLSB);
        PORT1 = PORT1 & 0xF8;
        write_digital(1, PORT1);
        PORT1 = PORT1 & 0xF7;
        write_digital(1, PORT1);
        PORT1 = (PORT1 | 0x08);
        write_digital(1, PORT1);
        UPPERS = ((value >> 8) & 0x0F) | (UPPERS & 0xF0);
        write_digital(0, ~UPPERS);
        PORT1 = (PORT1 & 0xF8) | 0x02;
        write_digital(1, PORT1);
        PORT1 = PORT1 & 0xF7;
        write_digital(1, PORT1);
        PORT1 = (PORT1 | 0x08);
        write_digital(1, PORT1);
}
```

```
/************************************************************************/
/*                                                                      */
void write_to_YTO(unsigned int value)
/*                                                                      */
/* This I/O routine writes the integer value to the YIG-tuned           */
/* oscillator's control port.                                           */
/*                                                                      */
/************************************************************************/

{
      YTOLSB = value;
      write_digital(0, ~YTOLSB);
      PORT1 = (PORT1 & 0xF8) | 0x01;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
      UPPERS = ((value >> 4) & 0xF0) | (UPPERS & 0x0F);
      write_digital(0, ~UPPERS);
      PORT1 = (PORT1 & 0xF8) | 0x02;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
}
```

```
/***********************************************************************/
/*                                                                     */
void write_to_VCO(unsigned int value)
/*                                                                     */
/* Similar to write_to_YTF, this routine programs the signal           */
/* generator's voltage-controlled oscillator.                          */
/*                                                                     */
/***********************************************************************/

{
      VCOMSB = (value >> 4) & 0xF0;
      write_digital(0, ~VCOMSB);
      PORT1 = (PORT1 & 0xF8) | 0x05;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
      VCOLSB = value;
      write_digital(0, ~VCOLSB);
      PORT1 = (PORT1 & 0xF8) | 0x04;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
}
```

```
/*************************************************************************/
/*                                                                       */
void write_switch(unsigned char value)
/*                                                                       */
/* Similar to write_to_YTF, this routine programs the signal             */
/* generator's internal selector switch.                                 */
/*                                                                       */
/*************************************************************************/

{
      CONTROL = value;
      write_digital(0, ~CONTROL);
      PORT1 = (PORT1 & 0xF8) | 0x03;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
}
```

```
/************************************************************************/
/*                                                                      */
void to_frequency(double frequency)
/*                                                                      */
/* This routine uses write_to_YTF, write_to_YTO, write_to_VCO, and      */
/* write_switch to command the signal generator to the specified        */
/* frequency.  To keep the filter aligned with the oscillator           */
/* frequencies, the global array cal_store[] is referenced.            */
/* cal_store[] contains a look-up table generated from an alignment     */
/* routine which gives the oscillator code which lines up with a        */
/* given filter code value.                                             */
/*                                                                      */
/************************************************************************/

{
      int  i;
      unsigned int YTF_code, YTO_code, VCO_code;
      unsigned char CONTROL_code;
      double   temp, templ;
      static double YTF_freq[6] = {1.0, 2.691, 4.394, 6.098, 7.798, 8.0};
      static double YTF_val[6] = {0.0, 994.0, 1989.0, 2983.0, 3977.0,
           4095.0};

      if ((frequency <= 8.0) && (frequency >= 2.0)){
           CONTROL_code = 0x02;
           write_switch(CONTROL_code);
           for (i = 0; i < 6; ++i){
                temp = frequency - YTF_freq[i];
                if (temp <= 0) break;
           }
           temp = YTF_val[i-1] + (frequency - YTF_freq[i-1])
                     / (YTF_freq[i] - YTF_freq[i-1])
                          * (YTF_val[i] - YTF_val[i-1]);
           templ = modf(temp, &temp);
           if (templ > 0.5) temp = temp + 1.0;
           YTF_code = (unsigned int) temp;
           write_to_YTF(YTF_code);
           write_to_YTO(calstore[YTF_code]);
      }
      else if ((frequency < 2.0) && (frequency >= 1.0)){
           CONTROL_code = 0x00;
           write_switch(CONTROL_code);
           temp = YTF_val[0] + (frequency - YTF_freq[0])
                     / (YTF_freq[1] - YTF_freq[0])
```

```
                              * (YTF_val[1] - YTF_val[0]);
              temp1 = modf(temp, &temp);
              if (temp1 > 0.5) temp = temp + 1.0;
              YTF_code = (unsigned int) temp;
              write_to_YTF(YTF_code);
              write_to_VCO(calstore[YTF_code]);
         }
}

/************************************************************************/
/*                                                                      */
void read_from_screen(int x, int y, int n, char *st)
/*                                                                      */
/* Uses DOS services to read data from the screen.  Returns a string   */
/* of length n read starting at position x,y on the screen.            */
/*                                                                      */
/************************************************************************/

{
    int i;
    union REGS a;

    gotoxy(x,y);
    for (i = 0; i < n; ++i){
          a.h.ah = 0x08;
          a.h.bh = 0x00;
          int86(0x10,&a,&a);
          st[i] = a.h.al;
          gotoxy(++x,y);
    }
}
```

```
/*********************************************************************/
/*                                                                   */
void update_data(int x, int y, struct inst_status *a, char ins_mode)
/*                                                                   */
/* Reads frequency and step size from screen and updates status data. */
/*                                                                   */
/*********************************************************************/

{
    char  st[8] = {' .',' .',' .',' .',' .',' .',' .',' .'};
    union REGS q;

    /* invisible cursor */

    q.h.ah = 0x01;
    q.h.ch = 0x20;
    q.h.cl = 0x00;
    int86(0x10,&q,&q);
    read_from_screen(19,10,6,st);
    if (sscanf(st," %lf",&(a->frequency)) != 0){
          if (a->frequency > 8.0) a->frequency = 8.0;
          if (a->frequency < 1.0) a->frequency = 1.0;
    }
    read_from_screen(19,11,6,st);
    if (sscanf(st," %lf",&(a->freq_step)) != 0){
          if (a->freq_step > 8.0) a->freq_step = 8.0;
          if (a->freq_step < 0.0) a->freq_step = 0.0;
    }

    /* restore cursor */

    gotoxy(x,y);
    if (ins_mode == 0){
        q.h.ah = 0x01;
      q.h.ch = 0x06;
      q.h.cl = 0x07;
      int86(0x10,&q,&q);
    }
    else{
      q.h.ah = 0x01;
      q.h.ch = 0x00;
      q.h.cl = 0x06;
      int86(0x10,&q,&q);
    }
}

/**********************************************************************/
/*                                                                    */
void initialize_screen(struct inst_status a, int x, int y)
/*                                                                    */
/* Sets up user screen in accordance with instrument status           */
/*                                                                    */
/**********************************************************************/

{
    char *int_time[4] = {" ", "10 ms", "100 ms", "1 s"};
    char *range[8] = {" ", "1 mV  ", "300 uV", "100 uV", "30 uV ", "10 uV  ",
                      "3 uV   ", "1 uV   "};
    char *bias_curs[2] = {"10 uA", "0.4 uA"};

    gotoxy(4,10); printf("Frequency:     %6.3lf GHz",a.frequency);
    gotoxy(4,11); printf("Freq. Inc.:    %6.3lf GHz",a.freq_step);
    gotoxy(4,12); printf("Step:          U/D        ");

    gotoxy(4,14); printf("Bias Curr.:    %s", bias_curs[a.bias_current]);
    gotoxy(4,15); printf("Int. Time:     %s", int_time[a.integ]);
    gotoxy(4,16); printf("Range:         %s", range[a.range]);
    gotoxy(x,y);
}
```

```
/***********************************************************************/
/*                                                                     */
void edit_line(int *curx, int *cury, char in_char, char insert_mode)
/*                                                                     */
/* Simple line editor for updating frequency and step size            */
/*                                                                     */
/***********************************************************************/

{
    int     i;
    char    temp[20];

    if ((*curx > 20) && (in_char == 0x08)){ /* Backspace */
        gettext(*curx, *cury, 24, *cury, temp);
        puttext(*curx - 1, *cury, 23, *cury, temp);
        gotoxy(24,*cury);
        putchar(' ');
        --(*curx);
    }
    else if (in_char == 0x7F){  /* Delete */
        gettext(*curx+1, *cury, 24, *cury, temp);
        puttext(*curx, *cury, 23, *cury, temp);
        gotoxy(24,*cury);
        putchar(' ');
    }
    else if (insert_mode == 0){
        putchar(in_char);
        *curx = wherex();
        if (*curx > 24) --(*curx);
    }
    else {
        gettext(*curx, *cury, 24, *cury, temp);
        puttext(*curx + 1, *cury, 24, *cury, temp);
        putchar(in_char);
        ++(*curx);
    }
}
```

```
#include <alloc.h>
#include <conio.h>
#include <dos.h>
#include <fcntl.h>
#include <graphics.h>
#include <io.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys\stat.h>
#include <time.h>
#define sqr(x) ((x)*(x))
#define pio12_pa 0x200
#define pio12_pb 0x201
#define pio12_pc 0x202
#define pio12_control 0x203
#define pclo_is_input 0x01
#define pclo_is_output 0x00
#define pb_is_input 0x02
#define pb_is_output 0x00
#define pb_pclo_is_mode_0 0x00
#define pb_pclo_is_mode_1 0x04
#define pchi_is_input 0x08
#define pchi_is_output 0x00
#define pa_is_input 0x10
#define pa_is_output 0x00
#define pa_pchi_is_mode_0 0x00
#define pa_pchi_is_mode_1 0x20
#define pa_pchi_is_mode_2 0x40
#define set_reset_mode 0x00
#define mode_set_active 0x80
#define number_of_channels 8
#define pi 3.14159265358979324
```

```
#define twopi 6.28318530717958648
#define v_l 299792458.0
#define   CW 0
#define SWEEP 1
#define CW_PULSE 2
#define SWEEP_PULSE 3
#define dt_2801_status_register 0x02ED
#define dt_2801_command_register 0x02ED
#define dt_2801_data_register 0x02EC
#define pi 3.14159265358979324
#define twopi 6.28318530717958648
#define v_l 299792458.0
#define STOP 0x0F
#define RESET 0x00
#define CLEAR_ERROR 0x01
#define SET_INT_CLK 0x03
#define SET_DIGITAL_OUTPUT 0x05
#define SET_A_D_PARAMS 0x0D
#define WRITE_DIG_IMM 0x07
#define READ_ERR 0x02
#define BOTH 0x02
#define DOR 0x01
#define DIF 0x02
#define READY 0x04
#define COMMAND 0x08
#define CERROR  0x80

struct inst_status {
    char      range, integ, bias_current;
    double    frequency, freq_step;
};

unsigned char YTOLSB, YTFLSB, UPPERS, CONTROL, PORT1, VCOLSB, VCOMSB;
int   YTFTWEAK = 0, MOD_ON = 0, ODD_NUM = 0;
double     FOFFSET = 0.0, DELTAF = 0.0;
int   *calstore;

void wait_microsec(int arg1);

void mode_set(char mode);

void set_gains_int(char g1, char g2, char int1, char bias);

void data_in();

void data_out();

void read_ads(char s_channel, char e_channel, unsigned int *out_vector);

void dt_error_check();

void dt_set_wait(unsigned char bytecode);

void dt_clear_wait(unsigned char bytecode);
void write_digital(unsigned char byte_select, unsigned char out_byte);

void dt_2801_set_up();

void write_to_YTF(unsigned int value);

void write_to_YTO(unsigned int value);

void write_to_VCO(unsigned int value);

void write_switch(unsigned char value);

void to_frequency(double frequency);

void read_from_screen(int x, int y, int n, char *st);

void update_data(int x, int y, struct inst_status *a, char ins_mode);

void initialize_screen(struct inst_status a, int x, int y);

void edit_line(int *curx, int *cury, char in_char, char insert_mode);
```

```
#include "caldefs.h"

/***********************************************************************/
/*                                                                     */
main()
/*                                                                     */
/* This program prompts for a frequency range and PDEU voltage range  */
/* and steps through the calibration procedure prompting the user at  */
/* each step.  No calibration calculations are performed here.  The   */
/* raw data from the calibration measurements are stored in a file.   */
/*                                                                     */
/***********************************************************************/

{
      char mode, out_val1, out_val2, out_val3, instring[20];
      char *range[8] = {" ", "1 mV ", "300 uV ", "100 uV ", "30 uV ",
           "10 uV ", "3 uV ", "1 uV "};
      int  i, j, k, l, OK, gain_val, nfreq;
      unsigned int out_vector[5];
      double   a, frequency, sum[5], sum_sq[5], sigma,
           offset[5], noise[5], ***big_matrix, **short_matrix,
           **open_matrix, **load_matrix, **meas_matrix, voltage,
           freq1, freq2, dfreq;
      char dummy;
      FILE *streamer, *streamer1;

      /* Open files and allocate space for data matrices */

      calstore = (int*) malloc(4096*sizeof(int));
      streamer = fopen("sweepcal.dat","rt");
      for (j = 0; j < 4096; ++j)
           fscanf(streamer, " %X", &(calstore[j]));
      fclose(streamer);

      big_matrix =
           (double ***) farmalloc((unsigned) Nslides*sizeof(double**));
      if (!big_matrix) nrerror("Memory allocation error.");
      for (i = 0; i < Nslides; ++i){
           big_matrix[i] =
              (double **) farmalloc((unsigned) Nfreqs*sizeof(double*));
           if (!(big_matrix[i])) nrerror("Memory allocation error.");
      }
      for (i = 0; i < Nslides; ++i)
           for (j = 0; j < Nfreqs; ++j){
                big_matrix[i][j] =
              (double *) farmalloc((unsigned) Ndiodes*sizeof(double));
           if (!(big_matrix[i][j])) nrerror("Memory allocation error.");
          }
     short_matrix =
          (double **) farmalloc((unsigned) Nfreqs*sizeof(double*));
     if (!short_matrix) nrerror("Memory allocation error.");
     for (i = 0; i < Nfreqs; ++i){
          short_matrix[i] =
          (double *) farmalloc((unsigned) Ndiodes*sizeof(double));
          if (!(short_matrix[i])) nrerror("Memory allocation error.");
     }
     open_matrix =
          (double **) farmalloc((unsigned) Nfreqs*sizeof(double*));
     if (!open_matrix) nrerror("Memory allocation error.");
     for (i = 0; i < Nfreqs; ++i){
          open_matrix[i] =
          (double *) farmalloc((unsigned) Ndiodes*sizeof(double));
          if (!(open_matrix[i])) nrerror("Memory allocation error.");
     }
     load_matrix =
          (double **) farmalloc((unsigned) Nfreqs*sizeof(double*));
     if (!load_matrix) nrerror("Memory allocation error.");
     for (i = 0; i < Nfreqs; ++i){
          load_matrix[i] =
          (double *) farmalloc((unsigned) Ndiodes*sizeof(double));
          if (!(load_matrix[i])) nrerror("Memory allocation error.");
     }
     meas_matrix =
          (double **) farmalloc((unsigned) Nfreqs*sizeof(double*));
     if (!meas_matrix) nrerror("Memory allocation error.");
     for (i = 0; i < Nfreqs; ++i){
          meas_matrix[i] =
          (double *) farmalloc((unsigned) Ndiodes*sizeof(double));
          if (!(meas_matrix[i])) nrerror("Memory allocation error.");
     }

     dt_2801_init(); /* initialize Data Translation board */
     data_out();     /* set PIO-12 interface for output to the PDEU */
     clrscr();
     OK = 0;         /* prompt for PDEU range setting */
     while (OK == 0){
          printf("Select preamp sensitivity:\n");
          printf("1 - 1mV full scale.\n");
          printf("2 - 300uV full scale.\n");
          printf("3 - 100uV full scale.\n");
          printf("4 - 30uV full scale.\n");
          printf("5 - 10uV full scale.\n");
          printf("6 - 3uV full scale.\n");
          printf("7 - 1uV full scale.\n");
          scanf(" %d", &gain_val);
          printf("You have selected %s full scale.\n", range[gain_val]);
          printf("Is this correct? (Y/N)\n");
          scanf("%s", instring);
          if ((instring[0] == 'Y') || (instring[0] == 'y'))
               OK = 1;
     }

     /* select gain codes for specified range setting */

     switch(gain_val){
          case 1 : set_gains_int(1,1,3,0);
               break;
          case 2 : set_gains_int(2,1,3,0);
               break;
          case 3 : set_gains_int(3,1,3,0);
               break;
          case 4 : set_gains_int(2,2,3,0);
               break;
          case 5 : set_gains_int(3,2,3,0);
               break;
          case 6 : set_gains_int(2,3,3,0);
               break;
          case 7 : set_gains_int(3,3,3,0);
               break;
     }
     data_in(); /* set PIO-12 interface for data output to PDEU */
     outportb(pio12_pc,0xC0);
     delay(50);           /* perform 1 A/D conversion to clear system */
     read_ads(3,7,out_vector);
     OK = 0;

     /* prompt for frequency range */

     while (OK == 0){
          printf("\n\nInput start frequency (GHz), stop frequency (GHz), and\n");
          printf("number of frequency points, separated by commas.\n");
          scanf(" %lf, %lf, %d", &freq1, &freq2, &nfreq);
          if (freq1 < 1.0) freq1 = 1.0;
          if (freq1 > 8.0) freq1 = 8.0;
          if (freq2 < 1.0) freq2 = 1.0;
          if (freq2 > 8.0) freq2 = 8.0;
          if (nfreq < 1) nfreq = 1;
          if (nfreq > 51) nfreq = 51;
          printf("You have selected\n");
          printf("Start frequency: %5.3lf GHz\n",freq1);
          printf("Start frequency: %5.3lf GHz\n",freq2);
          printf("Number of points: %d \n", nfreq);
          printf("Is this correct? (Y/N)\n");
          scanf("%s", instring);
          if ((instring[0] == 'Y') || (instring[0] == 'y'))
               OK = 1;
     }

     /* prompt for calibration file name and begin calibration */

     printf("Output file name?  ");
     scanf("%s",instring);
     streamer = fopen(instring,"wt");
     printf("Checking system.  Please wait.");
     YTFTWEAK = 0x0800;
     to_frequency(freq1);
     sleep(2);

     /* read system offsets and noise levels */

     for (i = 0; i < Ndiodes; ++i) sum[i] = sum_sq[i] = 0.0;
     for (i = 0; i < 20; ++i){
          delay(500);
          read_ads(3,7,out_vector);
          for (j = 0; j < 5; ++j){
               voltage = ((double)out_vector[j])/65535.0*10.0;
               sum[j] += voltage;
               sum_sq[j] += voltage * voltage;
          }
     }
     clrscr();
     for (i = 0; i < 5; ++i){
          offset[i] = sum[i] = sum[i] / 20.0;
          noise[i] = sigma
               = sqrt((sum_sq[i]-20.0*sum[i]*sum[i])/19.0);
          printf("Channel %1d: offset = %7.4lfV, s. dev. = %7.4lfV.\n",
                i, sum[i], sigma);
    }

    YTFTWEAK = 0x0000;
    to_frequency(freq1);

    /* main calibration loop */

    printf("\n\n");
    fflush(stdin);
    for (i = 0; i < Nslides; ++i){
         clrscr();
         printf("Slide short to position %2d.\n", i+1);
         printf("   Press RETURN when ready. .7");
         scanf("%c", &dummy);
         clrscr();
         for (j = 0; j < nfreq; ++j){
              frequency = freq1 + ((double) j) / ((double)(nfreq-1))
                   * (freq2 - freq1);
              to_frequency(frequency);
              gotoxy(4,10);
              printf("Frequency: %5.3lf GHz", frequency);
              sleep(2);
              read_ads(3,7,out_vector);
              for (k = 0; k < 5; ++k)
                   big_matrix[i][j][k] =
                        (double)out_vector[k]/65535.0*10.0
                             - offset[k];
         }
    }
    clrscr();
    printf("Place standard no. 1 at reference plane.\n");
    printf("   Press RETURN when ready. .7");
    scanf("%c", &dummy);
    clrscr();
    for (j = 0; j < nfreq; ++j){
         frequency = freq1 + ((double) j) / ((double)(nfreq-1))
              * (freq2 - freq1);
         to_frequency(frequency);
         gotoxy(4,10);
         printf("Frequency: %5.3lf GHz", frequency);
         sleep(2);
         read_ads(3,7,out_vector);
         for (k = 0; k < 5; ++k)
```

```
                    short_matrix[j][k] =
                         (double)out_vector[k]/65535.0*10.0
                              - offset[k];
          }
          clrscr();
          printf("Place standard no. 2 circuit at reference plane.\n");
          printf("   Press RETURN when ready. .7");
          scanf("%c", &dummy);
          clrscr();
          for (j = 0; j < nfreq; ++j){
               frequency = freq1 + ((double) j) / ((double)(nfreq-1))
                    * (freq2 - freq1);
               to_frequency(frequency);
               gotoxy(4,10);
               printf("Frequency: %5.3lf GHz", frequency);
               sleep(2);
               read_ads(3,7,out_vector);
               for (k = 0; k < 5; ++k)
                    open_matrix[j][k] =
                         (double)out_vector[k]/65535.0*10.0
                              - offset[k];
          }
          clrscr();
          printf("Place standard no. 3 at reference plane.\n");
          printf("   Press RETURN when ready. .7");
          scanf("%c", &dummy);
          clrscr();
          for (j = 0; j < nfreq; ++j){
               frequency = freq1 + ((double) j) / ((double)(nfreq-1))
                    * (freq2 - freq1);
               to_frequency(frequency);
               gotoxy(4,10);
               printf("Frequency: %5.3lf GHz", frequency);
               sleep(2);
               read_ads(3,7,out_vector);
               for (k = 0; k < 5; ++k)
                    load_matrix[j][k] =
                         (double)out_vector[k]/65535.0*10.0
                              - offset[k];
          }

          /* write data to output file */

          fprintf(streamer, "%d\n", gain_val);
```

```
    fprintf(streamer, "%lf\n", freq1);
    fprintf(streamer, "%lf\n", freq2);
    fprintf(streamer, "%d\n", nfreq);
    for (k = 0; k < Ndiodes; ++k)
         fprintf(streamer, "%9.5lf   ", offset[k]);
    fprintf(streamer,"\n");
    for (k = 0; k < Ndiodes; ++k)
         fprintf(streamer, "%9.5lf   ", noise[k]);
    fprintf(streamer,"\n");
    for (j = 0; j < nfreq; ++j){
         for (i = 0; i < Nslides; ++i){
              for (k = 0; k < Ndiodes; ++k)
                   fprintf(streamer, "%9.5lf   ",
                             big_matrix[i][j][k]);
              fprintf(streamer, "\n");
         }
         for (k = 0; k < Ndiodes; ++k)
              fprintf(streamer, "%9.5lf   ", short_matrix[j][k]);
         fprintf(streamer, "\n");
         for (k = 0; k < Ndiodes; ++k)
              fprintf(streamer, "%9.5lf   ", open_matrix[j][k]);
         fprintf(streamer, "\n");
         for (k = 0; k < Ndiodes; ++k)
              fprintf(streamer, "%9.5lf   ", load_matrix[j][k]);
         fprintf(streamer, "\n");
    }

    fclose(streamer);
    printf("\n\nCalibration complete.  File %s written.", instring);

    /* Free memory */

    for (i = (Nfreqs-1); i >= 0; --i) farfree(meas_matrix[i]);
    farfree(meas_matrix);
    for (i = (Nfreqs-1); i >= 0; --i) farfree(load_matrix[i]);
    farfree(load_matrix);
    for (i = (Nfreqs-1); i >= 0; --i) farfree(open_matrix[i]);
    farfree(open_matrix);
    for (i = (Nfreqs-1); i >= 0; --i) farfree(short_matrix[i]);
    farfree(short_matrix);
    for (i = (Nslides-1); i >=0; --i)
         for (j = (Nfreqs-1); j >= 0; --j)
              farfree(big_matrix[i][j]);
    for (i = (Nslides-1); i >=0; --i) farfree(big_matrix[i]);
```

```
      farfree(big_matrix);
      farfree(calstore);
}

/* End of main routine */

/************************************************************************/
/*                                                                      */
void wait_microsec(int arg1)
/*                                                                      */
/* Assembly language timing loop for short delays.  DOS and BIOS        */
/*                                                                      */
/************************************************************************/

{
asm   mov  cx,arg1
s1:
asm   nop
asm   loop s1
}

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

void interrupt (*oldfunc)();

/************************************************************************/
/*                                                                      */
void nrerror(char error_text[])
/*                                                                      */
/* General purpose error handling routine.                              */
/*                                                                      */
/************************************************************************/

{
      printf("%s\n",error_text);
      printf("...now exiting to system...\n");
      exit(1);
}
```

```
/**********************************************************************/
/*                                                                    */
void mode_set(char mode)
/*                                                                    */
/* Sets mode of programmable peripheral interface on PIO-12 board.   */
/*                                                                    */
/**********************************************************************/

{
      outportb(pio12_control, mode);
}
```

```
/************************************************************************/
/*                                                                      */
void dt_error_check()
/*                                                                      */
/* This routine is called whenever an error is reported by the Data    */
/* Translation board.  It queries the board, interprets the resulting */
/* error code, and prints an error message.                             */
/*                                                                      */
/************************************************************************/

{
      int  i;
      int  ready = 0;
      int  timeout = -32767;
      char a, b, reg;

      for (i = 0; i < 100; ++i){}
      outportb(dt_2801_command_register, READ_ERR);
      while (ready == 0){
           for (i = 0; i < 100; ++i){}
           reg = inportb(dt_2801_status_register);
           if ((reg & DOR) == DOR)
                ready = 1;
           else{
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
                     printf("during dt_error_check.\n");
                     ready = 1;
                }
           }
      }
      a = inportb(dt_2801_data_register);
      ready = 0;
      timeout = -32767;
      while (ready == 0){
           for (i = 0; i < 100; ++i){}
           reg = inportb(dt_2801_status_register);
           if ((reg & DOR) == DOR)
                ready = 1;
           else{
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
```

```
                              printf("during dt_error_check.\n");
                              ready = 1;
                         }
                    }
               }
               b = inportb(dt_2801_data_register);
               if ((a & 0x02) == 0x02)
                    printf("Command Overwrite Error\n");
               if ((a & 0x04) == 0x04)
                    printf("Clock Set Error\n");
               if ((a & 0x08) == 0x08)
                    printf("Digital Port Select Error\n");
               if ((a & 0x10) == 0x10)
                    printf("Digital Port Set Error\n");
               if ((a & 0x20) == 0x20)
                    printf("DAC Select Error\n");
               if ((a & 0x40) == 0x40)
                    printf("DAC Clock Error\n");
               if ((a & 0x80) == 0x80)
                    printf("DAC No. Conversions Value Error\n");
               if ((b & 0x01) == 0x01)
                    printf("A/D Channel Error\n");
               if ((b & 0x02) == 0x02)
                    printf("A/D Gain Error\n");
               if ((b & 0x04) == 0x04)
                    printf("A/D Clock Error\n");
               if ((b & 0x08) == 0x08)
                    printf("A/D Multiplexer Error\n");
               if ((b & 0x10) == 0x10)
                    printf("A/D No. Conversions Value Error\n");
               if ((b & 0x20) == 0x20)
                    printf("Data Where Command Expected Error\n");
          }
```

```
/***********************************************************************/
/*                                                                     */
void dt_set_wait(unsigned char bytecode)
/*                                                                     */
/* Part of the handshaking with the Data Translation board, this       */
/* routine waits until a particular byte pattern is set in the DT-     */
/* 2801's status register.                                             */
/*                                                                     */
/***********************************************************************/

{
      char ready, i, idle;
      int timeout;

      ready = 0;
      timeout = -32767;
      while (ready == 0){
           idle = 0;
           while (idle <= 100) ++idle;
           i = inportb(dt_2801_status_register);
           if ((i & 0x80) == 0x80) dt_error_check();
           if ((i & bytecode) == bytecode)
                ready = 1;
           else
           {
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
                     printf("during dt_set_wait.\n");
                     ready = 1;
                }
           }
      }
}
```

```
/***********************************************************************/
/*                                                                     */
void dt_set_wait_no_error(unsigned char bytecode)
/*                                                                     */
/* Same as dt_set_wait except this routine does not check for errors  */
/* from the DT-2801 board                                              */
/*                                                                     */
/***********************************************************************/

{
      char ready = 0, i, idle;
      int timeout;

      timeout = -32767;
      while (ready == 0){
           idle = 0;
           while (idle <= 100) ++idle;
           i = inportb(dt_2801_status_register);
           if ((i & bytecode) == bytecode)
                ready = 1;
           else
           {
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
                     printf("during dt_set_wait.\n");
                     ready = 1;
                }
           }
      }
}
```

```
/***********************************************************************/
/*                                                                     */
void dt_clear_wait(unsigned char bytecode)
/*                                                                     */
/* Part of the handshaking with the Data Translation board, this       */
/* routine waits until a particular byte pattern is cleared in the     */
/* DT-2801's status register.                                          */
/*                                                                     */
/***********************************************************************/

{
      char ready, i, idle;
      int timeout;

      ready = 0;
      timeout = -32767;
      while (ready == 0){
           idle = 0;
           while (idle <= 100) ++idle;
           i = inportb(dt_2801_status_register);
           if ((i & 0x80) == 0x80) dt_error_check();
           if ((i & bytecode) == 0)
                ready = 1;
           else
           {
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
                     printf("during dt_clear_wait.\n");
                     ready = 1;
                }
           }
      }
}
```

```
/*************************************************************************/
/*                                                                       */
void dt_2801_init()
/*                                                                       */
/* This routine initializes the Data Translation board.  It first       */
/* resets the board, then reads its identity and prints out the board */
/* type.  Then it executes the boards self-test, and sets both          */
/* digital ports for output.                                            */
/*                                                                       */
/*************************************************************************/

{
      unsigned char status, test_val, id, revno;
      int  i;

      status = inportb(dt_2801_status_register);
      if ((status & 0x70) != 0){
           printf("Illegal status register value\n");
           exit(1);
      }
      outportb(dt_2801_command_register, STOP); /* Stops execution of any */
      for (i = 0; i < 100; ++i){}               /* existing commands.     */
      status = inportb(dt_2801_data_register);  /* Dummy read to clear    */
      dt_set_wait_no_error(READY);              /* registers.             */
      outportb(dt_2801_command_register, RESET);/* Reset board.           */
      dt_set_wait_no_error(DOR);
      status = inportb(dt_2801_data_register);  /* Read board ID          */
      dt_set_wait(READY);
      id = status & 0xF0;
      revno = status & 0x0F;
      switch(id){                               /* Decode board ID number */
           case 0x00:    printf("DT-2801 board,");
                     break;
           case 0x10:    printf("DT-2805 board,");
                     break;
           case 0x20:    printf("DT-2808 board,");
                     break;
           case 0x30:    printf("DT-2808 board, with extender,");
                     break;
           case 0x50:    printf("DT-2801-A board,");
                     break;
           case 0x80:    printf("DT-2801/5716 board,");
                     break;
           case 0x90:    printf("DT-2805/5716 board,");
```

```
                    break;
          case 0xA0:    printf("DT-2818 board,");
     }
     /*printf(" firmware revision %2d.\n", revno);*/
     outportb(dt_2801_command_register, TEST);     /* Self test routine.  */
     for (i = 1; i < 256; ++i){                    /* Board should output */
          dt_set_wait(DOR);                    /* sequential values.  */
          test_val = inportb(dt_2801_data_register);
          status = inportb(dt_2801_status_register);
          if ((test_val != i) || ((status & 0x80) != 0)){
               printf("DT-2801 Failure in TEST Routine\n");
               exit(1);
          }
     }
     dt_set_wait(DOR);
     test_val = inportb(dt_2801_data_register);
     status = inportb(dt_2801_status_register);
     if ((test_val != 0) || ((status & 0x80) != 0)){
          printf("DT-2801 Failure in TEST Routine\n");
          exit(1);
     }
     outportb(dt_2801_command_register, STOP);      /* Stop self-test.   */
     for (i = 0; i < 100; ++i){}
     status = inportb(dt_2801_data_register);
     dt_set_wait_no_error(READY);
     outportb(dt_2801_command_register, RESET);    /* Reset board.       */
     dt_set_wait_no_error(DOR);
     status = inportb(dt_2801_data_register);
     dt_set_wait(READY);
     outportb(dt_2801_command_register,SET_DIGITAL_OUTPUT); /* Set up    */
     dt_clear_wait(DIF);                      /* both digital ports for  */
     outportb(dt_2801_data_register,BOTH);    /* output.                 */
     dt_clear_wait(DIF);
     dt_set_wait(READY);
}
```

```
/***********************************************************************/
/*                                                                     */
void write_digital(unsigned char byte_select, unsigned char out_byte)
/*                                                                     */
/* This I/O routine writes the byte out_byte to the DT-2801 parallel  */
/* port selected by byte_select.  The value of byte_select can be 1   */
/* or 0 since the DT-2801 has only 2 ports.                           */
/*                                                                     */
/***********************************************************************/

{
      int  i;

      outportb(dt_2801_command_register, WRITE_DIG_IMM);
      dt_clear_wait(DIF);
      outportb(dt_2801_data_register, byte_select);
      dt_clear_wait(DIF);
      outportb(dt_2801_data_register, out_byte);
      dt_clear_wait(DIF);
      dt_set_wait(READY);
}

/***********************************************************************/
/*                                                                     */
void write_to_YTF(unsigned int value)
/*                                                                     */
/* This I/O routine writes the integer value to the YIG-tuned         */
/* filter's control port.                                              */
/*                                                                     */
/***********************************************************************/

{
      unsigned char a;

      value += YTFTWEAK;  /* Add filter offset value (YTFTWEAK) to  */
      YTFLSB = value;     /* input value and store resulting YTF    */
      write_digital(0, ~YTFLSB);  /* code in global variables       */
      PORT1 = PORT1 & 0xF8;  /* Note: port writes are complemented  */
      write_digital(1, PORT1); /* due to logical inversion in       */
      PORT1 = PORT1 & 0xF7;    /* electronics. Write LSB first.     */
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08); /* Toggle CLOCK bit in control register. */
      write_digital(1, PORT1);
      UPPERS = ((value >> 8) & 0x0F) | (UPPERS & 0xF0);
      write_digital(0, ~UPPERS); /* Compute and write combination of */
      PORT1 = (PORT1 & 0xF8) | 0x02; /* upper nybbles.               */
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);  /* Toggle CLOCK to latch values in  */
      PORT1 = (PORT1 | 0x08);   /* signal generator registers.      */
      write_digital(1, PORT1);
}
```

```
/***********************************************************************/
/*                                                                     */
void write_to_YTO(unsigned int value)
/*                                                                     */
/* This I/O routine writes the integer value to the YIG-tuned          */
/* oscillator's control port.                                          */
/*                                                                     */
/***********************************************************************/

{
        YTOLSB = value;        /* Store value in global variable. */
        write_digital(0, ~YTOLSB);
        PORT1 = (PORT1 & 0xF8) | 0x01;
        write_digital(1, PORT1); /* Clock value into signal generator */
        PORT1 = PORT1 & 0xF7;    /* registers. */
        write_digital(1, PORT1);
        PORT1 = (PORT1 | 0x08);
        write_digital(1, PORT1);
        UPPERS = ((value >> 4) & 0xF0) | (UPPERS & 0x0F);
        write_digital(0, ~UPPERS);
        PORT1 = (PORT1 & 0xF8) | 0x02;  /* Calculate and write combination */
        write_digital(1, PORT1);        /* of upper nybbles. */
        PORT1 = PORT1 & 0xF7;
        write_digital(1, PORT1);
        PORT1 = (PORT1 | 0x08);
        write_digital(1, PORT1);
}
```

```
/**************************************************************************/
/*                                                                        */
void write_to_VCO(unsigned int value)
/*                                                                        */
/* Similar to write_to_YTF, this routine programs the signal              */
/* generator's voltage-controlled oscillator.                             */
/*                                                                        */
/**************************************************************************/

{
      VCOMSB = (value >> 4) & 0xF0;
      write_digital(0, ~VCOMSB);
      PORT1 = (PORT1 & 0xF8) | 0x05;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
      VCOLSB = value;
      write_digital(0, ~VCOLSB);
      PORT1 = (PORT1 & 0xF8) | 0x04;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
}
```

```
/**********************************************************************/
/*                                                                    */
void write_switch(unsigned char value)
/*                                                                    */
/* Similar to write_to_YTF, this routine programs the signal         */
/* generator's internal selector switch.                             */
/*                                                                    */
/**********************************************************************/

{
      CONTROL = value;
      write_digital(0, ~CONTROL);
      PORT1 = (PORT1 & 0xF8) | 0x03;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
}
```

```
/************************************************************************/
/*                                                                      */
void to_frequency(double frequency)
/*                                                                      */
/* This routine uses write_to_YTF, write_to_YTO, write_to_VCO, and     */
/* write_switch to command the signal generator to the specified       */
/* frequency.  To keep the filter aligned with the oscillator          */
/* frequencies, the global array cal_store[] is referenced.           */
/* cal_store[] contains a look-up table generated from an alignment    */
/* routine which gives the oscillator code which lines up with a       */
/* given filter code value.                                            */
/*                                                                      */
/************************************************************************/

{
      int  i;
      unsigned int YTF_code, YTO_code, VCO_code;
      unsigned char CONTROL_code;
      double    temp, temp1;
      static double YTF_freq[6] = {1.0, 2.691, 4.394, 6.098, 7.798, 8.0};
      static double YTF_val[6] = {0.0, 994.0, 1989.0, 2983.0, 3977.0,
           4095.0};

      /* The arrays YTF_freq[] and YTF_val[] hold the frequency    */
      /* calibration of the YIG-tuned filter.                      */

      if ((frequency <= 8.0) && (frequency >= 2.0)){
           CONTROL_code = 0x02;        /* Select YTO for 2<f<8GHz */
           write_switch(CONTROL_code);
           for (i = 0; i < 6; ++i){
                temp = frequency - YTF_freq[i];
                if (temp <= 0) break;
           }
           /* Calculate digital code for YIG-tuned filter. */
           temp = YTF_val[i-1] + (frequency - YTF_freq[i-1])
                     / (YTF_freq[i] - YTF_freq[i-1])
                          * (YTF_val[i] - YTF_val[i-1]);
           temp1 = modf(temp, &temp);
           if (temp1 > 0.5) temp = temp + 1.0;
           YTF_code = (unsigned int) temp;
           write_to_YTF(YTF_code);
           write_to_YTO(calstore[YTF_code]); /* Get YTO code from     */
                                             /* look-up table.        */
      }
```

```
        else if ((frequency < 2.0) && (frequency >= 1.0)){
             CONTROL_code = 0x00;        /* Select VCO for 1<f<2GHz   */
             write_switch(CONTROL_code);
             temp = YTF_val[0] + (frequency - YTF_freq[0])
                       / (YTF_freq[1] - YTF_freq[0])
                            * (YTF_val[1] - YTF_val[0]);
             temp1 = modf(temp, &temp);
             if (temp1 > 0.5) temp = temp + 1.0;
             YTF_code = (unsigned int) temp;
             write_to_YTF(YTF_code);
             write_to_VCO(calstore[YTF_code]);
        }
}
```

```
/*********************************************************************/
/*                                                                   */
void set_gains_int(char g1, char g2, char int1, char bias)
/*                                                                   */
/* This routine sets the gains of the various programmable ampli-    */
/* fiers in the network analyzer's post-detection electronics.  It   */
/* also sets the integration time and the bias current.              */
/* Communication with the post-detection electronics is through      */
/* the PIO-12 24-bit parallel interface card.                        */
/*                                                                   */
/*********************************************************************/

{
        outportb(pio12_pb, 0x14); /* Select gain1/gain2/bias register */
asm     jmp  $+2          /* I/O delay to for PIO-12 card */
        outportb(pio12_pa,
              ~((0x01 << (g1-1)) | (0x08 << (g2-1)) |
                                      ((bias << 7) & 0x80)));
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2
        outportb(pio12_pc, 0x80); /* Clock value into registers */
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2

        outportb(pio12_pb, 0x15); /* Select integration time register */
asm     jmp  $+2
        outportb(pio12_pa, ~(0x01 << (int1-1)));
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2
        outportb(pio12_pc, 0x80);  /* Clock value into register */
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2
}
```

```
/************************************************************************/
/*                                                                      */
void data_in()
/*                                                                      */
/* Set PIO-12 board up for input on one of its 8-bit ports.  Keep      */
/* other 2 ports in output mode for control.                           */
/*                                                                      */
/************************************************************************/

{
        char mode;

        outportb(pio12_pa, 0x00);
asm     jmp  $+2
        outportb(pio12_pb, 0x00);
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2

        mode = pa_is_input | pb_is_output | pclo_is_output
             | pchi_is_output | pa_pchi_is_mode_0 | pb_pclo_is_mode_0
                  | mode_set_active;
        mode_set(mode);
}

/************************************************************************/
/*                                                                      */
void data_out()
/*                                                                      */
/* Set PIO-12 board up for output on all of its 8-bit ports.           */
/*                                                                      */
/************************************************************************/

{
        char mode;

        mode = pa_is_output | pb_is_output | pclo_is_output
             | pchi_is_output | pa_pchi_is_mode_0 | pb_pclo_is_mode_0
                  | mode_set_active;
        mode_set(mode);
}
```

```
/***********************************************************************/
/*                                                                     */
void read_ads(char s_channel, char e_channel, unsigned int *out_vector)
/*                                                                     */
/* Uses the 16-bit A/D converter in the post-detection electronics    */
/* to read the analog outputs of the various synchronous detectors.   */
/* s_channel is first channel to be read, e_channel is last.  Array   */
/* of converted values is returned in out_vector.                     */
/*                                                                     */
/***********************************************************************/

{
        char byte1[2], i;
        unsigned int   *temp;

        temp = (unsigned int *) byte1;

        outportb(pio12_pc, 0xC0);
        delay(2);
        outportb(pio12_pc, 0xC8); /* Enable A/D conversion */
        wait_microsec(100);
        for (i = s_channel; i <= e_channel; ++i){
                outportb(pio12_pb, 0x08 | (i & 0x07)); /* Select analog */
asm     jmp     $+2;                                    /* channel.      */
                outportb(pio12_pc, 0x88);  /* Clock channel select into */
asm     jmp     $+2;                       /* register.                 */
                outportb(pio12_pc, 0xC8);
                wait_microsec(10);        /* S/H settling time */
                outportb(pio12_pc, 0xE8); /* Start A/D conversion */
asm     jmp     $+2;
                outportb(pio12_pc, 0xC8);
                wait_microsec(100);
                outportb(pio12_pb, 0x18);
asm     jmp     $+2;
                byte1[0] = inportb(pio12_pa); /* read low byte */
asm     jmp     $+2;
                outportb(pio12_pb, 0x19);
asm     jmp     $+2;
                byte1[1] = inportb(pio12_pa); /* read high byte */
                out_vector[i-s_channel] = ~(*temp); /* A/D output is */
                        /* complementary offset binary.  Therefore invert */
                        /* results.  */
        }
        outportb(pio12_pc, 0xC0);
}
```

```
#include <alloc.h>
#include <conio.h>
#include <dos.h>
#include <fcntl.h>
#include <graphics.h>
#include <io.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys\stat.h>
#include <time.h>
#define sqr(x) ((x)*(x))
#define pio12_pa 0x200
#define pio12_pb 0x201
#define pio12_pc 0x202
#define pio12_control 0x203
#define pclo_is_input 0x01
#define pclo_is_output 0x00
#define pb_is_input 0x02
#define pb_is_output 0x00
#define pb_pclo_is_mode_0 0x00
#define pb_pclo_is_mode_1 0x04
#define pchi_is_input 0x08
#define pchi_is_output 0x00
#define pa_is_input 0x10
#define pa_is_output 0x00
#define pa_pchi_is_mode_0 0x00
#define pa_pchi_is_mode_1 0x20
#define pa_pchi_is_mode_2 0x40
#define set_reset_mode 0x00
#define mode_set_active 0x80
#define pi 3.14159265358979324
#define twopi 6.28318530717958648
#define v_l 299792458.0
#define    CW 0
#define SWEEP 1
#define CW_PULSE 2
#define SWEEP_PULSE 3
#define dt_2801_status_register 0x02ED
#define dt_2801_command_register 0x02ED
#define dt_2801_data_register 0x02EC
#define STOP 0x0F
#define RESET 0x00
```

```
#define CLEAR_ERROR 0x01
#define SET_INT_CLK 0x03
#define SET_DIGITAL_OUTPUT 0x05
#define TEST 0x0B
#define SET_A_D_PARAMS 0x0D
#define WRITE_DIG_IMM 0x07
#define READ_ERR 0x02
#define BOTH 0x02
#define DOR 0x01
#define DIF 0x02
#define READY 0x04
#define COMMAND 0x08
#define CERROR  0x80
#define Nslides 20
#define Nfreqs 51
#define Ndiodes 5

struct sweeper_status {
    int    mode;
    double    frequency, freq_step, start_freq, stop_freq, sweep_time;
};

unsigned char YTOLSB, YTFLSB, UPPERS, CONTROL, PORT1, VCOLSB, VCOMSB;
int   YTFTWEAK = 0, MOD_ON = 0, ODD_NUM = 0;
double     FOFFSET = 0.0, DELTAF = 0.0;
int   *calstore;

void wait_microsec(int arg1);

void interrupt (*oldfunc)();;

void mode_set(char mode);

void nrerror(char error_text[]);

void dt_error_check();

void dt_set_wait(unsigned char bytecode);

void dt_set_wait_no_error(unsigned char bytecode);

void dt_clear_wait(unsigned char bytecode);

void dt_2801_init();
void write_digital(unsigned char byte_select, unsigned char out_byte);

void write_to_YTF(unsigned int value);

void write_to_YTO(unsigned int value);

void write_to_VCO(unsigned int value);

void write_switch(unsigned char value);

void to_frequency(double frequency);

void set_gains_int(char g1, char g2, char int1, char bias);

void data_in();

void data_out();

void read_ads(char s_channel, char e_channel, unsigned int *out_vector);
```

```
#include "measure.h"

/************************************************************************/
/*                                                                      */
main()
/*                                                                      */
/* This program performs a single measurement sweep using the          */
/* frequency range and calibration data stored by the program          */
/* "calib.exe."  All calibration and measurement calculations, as      */
/* well as models for the calibration standards, are included here.    */
/*                                                                      */
/************************************************************************/

{
      double *b, *w, chisq, **u, **u_out, **v, *x1, *x2, *y, *diode_volts,
           frequency, *sig, atten, center_mag, zeta, AA, BB, CC, DD,
           EE, FF, GG, HH, II, ea, eb, ec, rho, temp, alpha, beta,
           gamma1,   delta, epsilon;
      freq_node *b1, *a1;  /* The data structures freq_node and cal_node, */
      cal_node *b2, *a2;   /* defined in measure.h, hold all the required */
                           /* calibration constants for the analyzer at a */
                           /* given frequency.                            */
      int  i, j, k, Nsamp, diodes[3], freq_loop, gain_val, nfreq;
      complex   gamma, gamma2;
      double    *mean, *standard_deviation, *sum_x, *sum_x_2, *noise, *offset;
      double    **working_matrix, **working_vector, P3, P4, P5, P6, freq1,
           freq2, dfreq;
      FILE *streamer, *streamer1;
      char dummy, instring[20], instring1[20];
      unsigned int out_vector[5];

      calstore = (int*) malloc(4096*sizeof(int));
      streamer = fopen("sweepcal.dat","rt");
      for (j = 0; j < 4096; ++j)
           fscanf(streamer, " %X", &(calstore[j]));
      fclose(streamer);

      /* Allocate memory */

      b1 = NULL;
      a1 = b1 = add_freq_node(b1);
      get_complex_mem();
      u = dmatrix(1,Nslides,1,9);
      u_out = dmatrix(1,Nslides,1,9);
```

```
        v = dmatrix(1,9,1,9);
        b = vector(1,Nslides);
        w = vector(1,9);
        x1 = vector(1,9);
        noise = vector(1,5);
        offset = vector(1,5);
        mean = vector(1,5);
        standard_deviation = vector(1,5);
        sum_x = vector(1,5);
        sum_x_2 = vector(1,5);
        sig = vector(1,40);
        diode_volts = vector(0,4);

        /* Check for existence of measure.cfg.  If it exists, read default */
        /* calibration file stored from previous run.                      */

        streamer = fopen("measure.cfg","rt");
        if (streamer){
             fscanf(streamer," %s", instring);
             printf("\n\nCalibration file name? (%s) ",instring);
             dummy = getch();

             /* If input is not a <CR>, read new cal file name. */

             if (dummy != 13){
                  ungetch(dummy);
                  scanf("%s",instring1);
                  strcpy(instring1,instring);
             }
        }
        else{   /* If measure.cfg does not exist, read new cal file name.*/
             printf("\n\nCalibration file name? ");
             scanf("%s",instring);
        }
        fclose(streamer);

        /* Write cal file name for use next time. */

        streamer = fopen("measure.cfg","wt");
        fprintf(streamer, "%s\n", instring);
        fclose(streamer);
        streamer = fopen(instring,"rt");
        if (!streamer){
             printf("Calibration file not found.");
```

```
          printf("Exiting to system...");
          exit(1);
     }

     /* Repeat above procedure for output filename. */

     streamer1 = fopen("measure1.cfg","rt");
     if (streamer1){
          fscanf(streamer1," %s", instring);
          printf("\n\nOutput file name? (%s) ",instring);
          dummy = getch();
          if (dummy != 13){
               ungetch(dummy);
               scanf("%s",instring1);
               strcpy(instring1,instring);
          }
     }
     else{
          printf("\n\nOutput file name? ");
          scanf("%s",instring);
     }
     fclose(streamer1);
     streamer1 = fopen("measure1.cfg","wt");
     fprintf(streamer1, "%s\n", instring);
     fclose(streamer1);
     streamer1 = fopen(instring,"wt");

     /* Read calibration file. */

     fscanf(streamer, " %d", &gain_val);
     fscanf(streamer, " %lf", &freq1);
     fscanf(streamer, " %lf", &freq2);
     fscanf(streamer, " %d", &nfreq);

     dt_2801_init();          /* Initialized DT-2801 board. */
     YTFTWEAK = 0x0000;
     to_frequency(freq1);     /* Go to first frequency. */
     data_out();              /* Set PIO-12 interface for output to PDEU */
     switch(gain_val){        /* Set system gain per value stored in     */
                         /* cal file.                                */
          case 1 : set_gains_int(1,1,3,0);
               break;
          case 2 : set_gains_int(2,1,3,0);
               break;
          case 3 : set_gains_int(3,1,3,0);
               break;
          case 4 : set_gains_int(2,2,3,0);
               break;
          case 5 : set_gains_int(3,2,3,0);
               break;
          case 6 : set_gains_int(2,3,3,0);
               break;
          case 7 : set_gains_int(3,3,3,0);
               break;
     }
     data_in();         /* Set PIO-12 for input from PDEU */
     outportb(pio12_pc,0xC0);
     delay(50);         /* Do one A/D conversion to clear system */
     read_ads(3,7,out_vector);

     /* Continue reading cal file. */

     for (i = 1; i <= Ndiodes; ++i) fscanf(streamer, " %lf", &(offset[i]));
     for (i = 1; i <= Ndiodes; ++i) fscanf(streamer, " %lf", &(noise[i]));

     /* Begin measurement sweep, reading values from cal file as needed. */

     fflush(stdin);
     printf("\n\nPlace device under test at reference plane.\n");
     printf("   Press RETURN when ready. .7");
     scanf("%c", &dummy);

     clrscr();
     for (freq_loop = 0; freq_loop < nfreq; ++freq_loop){
          if (freq_loop != 0) b1 = add_freq_node(a1);

          /* Data structure for cal and measurement data is a linked */
          /* list of records of type freq_node.  The routine         */
          /* add_freq_node adds to this list.                        */

          b1->frequency = freq1 + ((double) freq_loop) /
                                  ((double)(nfreq-1))
                    * (freq2 - freq1);
          to_frequency(b1->frequency);

          /* Calculate frequency and indicate it on the CRT as the  */
          /* sweep progresses. */
```

```
gotoxy(4,10);
printf("Frequency: %5.3lf GHz", b1->frequency);
sleep(2);

/* Read sliding short data from file. */

for (k = 1; k <= Nslides; ++k){
     b2 = add_cal_node(b1);
     for (i = 0; i < Ndiodes; ++i)
          fscanf(streamer," %lf",&(b2->diode_volts[i]));
     permute(b2->diode_volts, b1->frequency);
     P4 = b2->diode_volts[0];
     P3 = b2->diode_volts[1];
     P5 = b2->diode_volts[2];

     /* Fill 5X5 array for 5-port to 4-port conversion. */

     u[k][1] = P3*P3/P4/P4;
     u[k][2] = P3*P5/P4/P4;
     u[k][3] = P5*P5/P4/P4;
     u[k][4] = P3/P4;
     u[k][5] = P5/P4;
     b[k] = -1.0;
}

/* Read Short/Open/Load data from file */

for (i = 0; i < Ndiodes; ++i)
     fscanf(streamer, " %lf", &(b1->z1_volts[i]));
permute(b1->z1_volts, b1->frequency);

     /* The routine permute() rearranges the array of */
     /* measured voltages as a function of frequency  */
     /* such that the three voltages to be used at a  */
     /* given frequency are in array elements 0,1,    */
     /* and 2.                                        */

for (i = 0; i < Ndiodes; ++i)
     fscanf(streamer, " %lf", &(b1->z2_volts[i]));
permute(b1->z2_volts, b1->frequency);
for (i = 0; i < Ndiodes; ++i)
     fscanf(streamer, " %lf", &(b1->z3_volts[i]));
permute(b1->z3_volts, b1->frequency);
```

```
/* Read measured data from network analyzer. */

read_ads(3,7,out_vector);
for (i = 0; i < Ndiodes; ++i)
     b1->meas_volts[i] =
          (double)out_vector[i]/65535.0*10.0
               - offset[i];
permute(b1->meas_volts, b1->frequency);

/* Perform 5-port to 4-port reduction as described in   */
/* text of final report.  First step is to solve 5X5    */
/* matrix by singular value decomposition to yield      */
/* estimates of coefficients of best-fit ellipse.       */

svdcmp(u,Nslides,5,w,v);
temp = 0.0;
for (i = 1; i <= 5; ++i)
     if (fabs(w[i]) > temp) temp = w[i];
for (i = 1; i <= 5; ++i)
     if (fabs(w[i]) < temp/10000.0) w[i] = 0.0;

svbksb(u,w,v,Nslides,5,b,x1);

AA = x1[1];
BB = x1[2]/2.0;
CC = x1[3];
DD = x1[4]/2.0;
EE = x1[5]/2.0;

/* AA...EE are coefficients of best-fit ellipse.  Use these */
/* to find calibration constants as described in text of    */
/* final report.                                             */

alpha = (BB*DD-AA*EE)/(AA*CC-BB*BB);
beta = (DD*EE-BB)/(AA*CC-BB*BB);
gamma1 = (BB*EE-DD*CC)/(AA*CC-BB*BB);
temp = (AA-DD*DD)/(AA*CC-BB*BB);
delta = -sqrt(temp);
temp = (CC-EE*EE)/(AA*CC-BB*BB);
epsilon = -sqrt(temp);
ea = (alpha-delta)*(gamma1+epsilon)/2.0/(alpha+delta);
eb = (gamma1-epsilon)/2.0;
ec = (beta-epsilon*delta)/(alpha+delta);
zeta = (gamma1+epsilon)/(alpha+delta);
```

```
          b1->center_r[2] = b1->center_mag[2] = sqrt(ec);
          b1->center_i[2] = b1->theta[2] = 0.0;
          b1->scale[2] = zeta;
     }

     /* Perform 4-port (Short/Open/Load) calibration. */

     four_port_cal(a1);
     b1 = a1;
     clrscr();

     /* Find reflection coefficients from measured data. */

     for (freq_loop = 0; freq_loop < nfreq; ++freq_loop){
          find_S(b1);

          /* Write measured data to output file. */

          printf("%9.5lf   %9.5lf   %9.5lf\n",
               b1->frequency, c_abs(&(b1->S[0])),
                    atan2(b1->S[0].y, b1->S[0].x)/pi*180.0);
          fprintf(streamer1, "%9.5lf   %9.5lf   %9.5lf\n",
               b1->frequency, c_abs(&(b1->S[0])),
                    atan2(b1->S[0].y, b1->S[0].x)/pi*180.0);
          b1 = b1->next_freq_node;
     }

     /* Close files and free memory. */

     fclose(streamer);
     fclose(streamer1);
     free_vector(diode_volts,0,4);
     free_vector(sig,1,40);
     free_vector(sum_x_2,1,5);
     free_vector(sum_x,1,5);
     free_vector(standard_deviation,1,5);
     free_vector(mean,1,5);
     free_vector(offset,1,5);
     free_vector(noise,1,5);
     free_vector(x1,1,9);
     free_vector(w,1,9);
     free_vector(b,1,Nslides);
     free_dmatrix(v,1,9,1,9);
```

```
        free_dmatrix(u_out,1,Nslides,1,9);
        free_dmatrix(u,1,Nslides,1,9);
        free_complex_mem();
}

/* End of main routine */

/************************************************************************/
/*                                                                      */
void nrerror(char error_text[])
/*                                                                      */
/* General purpose error handling routine.                              */
/*                                                                      */
/************************************************************************/

{
        fprintf(stderr,"Numerical Recipes run-time error...\n");
        fprintf(stderr,"%s\n",error_text);
        fprintf(stderr,"...now exiting to system...\n");
        exit(1);
}
```

```
/**********************************************************************/
/*                                                                    */
void get_complex_mem(void)
/*                                                                    */
/* Allocates memory for complex arithmetic and initializes pointers.  */
/*                                                                    */
/**********************************************************************/

{
      complex_pointer
           = complex_base
                = farmalloc(500     * sizeof(complex));
      if (complex_base == NULL){
           printf("Error in get_complex_mem -- not enough memory\n");
           exit(1);
      }
      c_mat_pointer
           = c_mat_base
                = farmalloc(500 * sizeof(complex_mat));
      if (c_mat_base == NULL){
           printf("Error in get_complex_mem -- not enough memory\n");
           exit(1);
      }
      --complex_pointer;
      --complex_base;
      --c_mat_pointer;
      --c_mat_base;
}
```

```
/***********************************************************************/
/*                                                                     */
complex *co(double x, double y)
/*                                                                     */
/* Create a complex number with real part x and imaginary part y.     */
/*                                                                     */
/***********************************************************************/

{
      complex *k;

      k = next_complex_value();
      k->x = x;
      k->y = y;
      return(k);
}

/***********************************************************************/
/*                                                                     */
complex *su(complex *a, complex *b)
/*                                                                     */
/* Add two complex numbers.                                            */
/*                                                                     */
/***********************************************************************/
{
      complex *k;

      k = next_complex_value();
      k->x = a->x + b->x;
      k->y = a->y + b->y;
      return(k);
}
```

```
/**********************************************************************/
/*                                                                    */
complex *di(complex *a, complex *b)
/*                                                                    */
/* Subtract two complex numbers.                                      */
/*                                                                    */
/**********************************************************************/
{
      complex *k;

      k = next_complex_value();
      k->x = a->x - b->x;
      k->y = a->y - b->y;
      return(k);
}

/**********************************************************************/
/*                                                                    */
complex *pr(complex *a, complex *b)
/*                                                                    */
/* Multiply two complex numbers.                                      */
/*                                                                    */
/**********************************************************************/
{
      complex *k;

      k = next_complex_value();
      k->x = a->x * b->x - a->y * b->y;
      k->y = a->x * b->y + a->y * b->x;
      return(k);
}
```

```
/**********************************************************************/
/*                                                                    */
complex *qu(complex *a, complex *b)
/*                                                                    */
/* Divide two complex numbers.                                        */
/*                                                                    */
/**********************************************************************/
{
       complex *k;
       double r,den;

       k = next_complex_value();
       if (fabs(b->x) >= fabs(b->y)) {
            r=b->y/b->x;
            den=b->x+r*b->y;
            k->x=(a->x+r*a->y)/den;
            k->y=(a->y-r*a->x)/den;
       } else {
            r=b->x/b->y;
            den=b->y+r*b->x;
            k->x=(a->x*r+a->y)/den;
            k->y=(a->y*r-a->x)/den;
       }
       return k;
}
```

```
/***********************************************************************/
/*                                                                     */
double c_abs(complex *a)
/*                                                                     */
/* Return absolute value of a complex number.                          */
/*                                                                     */
/***********************************************************************/
{
      double x,y,ans,temp;

      x=fabs(a->x);
      y=fabs(a->y);
      if (x == 0.0)
           ans=y;
      else if (y == 0.0)
           ans=x;
      else if (x > y) {
           temp=y/x;
           ans=x*sqrt(1.0+temp*temp);
      } else {
           temp=x/y;
           ans=y*sqrt(1.0+temp*temp);
      }
      return ans;
}
```

```
/***********************************************************************/
/*                                                                     */
complex *c_sqrt(complex *a)
/*                                                                     */
/* Square root of a complex number.                          */
/*                                                                     */
/***********************************************************************/
{
      complex *c;
      double x,y,w,r;

      c = next_complex_value();
      if ((a->x == 0.0) && (a->y == 0.0)) {
           c->x=0.0;
           c->y=0.0;
           return c;
      } else {
           x=fabs(a->x);
           y=fabs(a->y);
           if (x >= y) {
                r=y/x;
                w=sqrt(x)*sqrt(0.5*(1.0+sqrt(1.0+r*r)));
           } else {
                r=x/y;
                w=sqrt(y)*sqrt(0.5*(r+sqrt(1.0+r*r)));
           }
           if (a->x >= 0.0) {
                c->x=w;
                c->y=a->y/(2.0*w);
           } else {
                c->y=(a->y >= 0) ? w : -w;
                c->x=a->y/(2.0*c->y);
           }
           return c;
      }
}
```

```
/**********************************************************************/
/*                                                                    */
complex    *rc_mul(double x, complex *a)
/*                                                                    */
/* Multiply a real and a complex number.                              */
/*                                                                    */
/**********************************************************************/
{
      complex *c;

      c = next_complex_value();
      c->x=x*a->x;
      c->y=x*a->y;
      return c;
}

/**********************************************************************/
/*                                                                    */
complex *c_exp(complex *z)
/*                                                                    */
/* Exponentiate a complex number.                                     */
/*                                                                    */
/**********************************************************************/
{
      complex *c;
      double   temp;

      temp = exp(z->x);
      c = co(temp*cos(z->y), temp*sin(z->y));
      return c;
}
```

```
/**********************************************************************/
/*                                                                    */
cal_node *add_cal_node(freq_node *a)
/*                                                                    */
/* Add one node to the linked list of calibration data records.      */
/*                                                                    */
/**********************************************************************/
{
      cal_node  *x, *y;

      x = a->first_cal_node;
      if (x == NULL){
           x = a->first_cal_node = farmalloc(cal_size);
           if (!x) nrerror("Memory allocation error.\n");
      }
      else{
           while (x->next_cal_node != NULL) x = x->next_cal_node;
           x = x->next_cal_node = farmalloc(cal_size);
           if (!x) nrerror("Memory allocation error.\n");
      }
      x->next_cal_node = NULL;
      return(x);
}
```

```
/**********************************************************************/
/*                                                                    */
freq_node *add_freq_node(freq_node *a)
/*                                                                    */
/* Add one node to the linked list of calibration and measurement     */
/* data records.                                                      */
/*                                                                    */
/**********************************************************************/
{
      freq_node *x, *y;
      unsigned long page1, page2;

      if (a == NULL){
           x = farmalloc(freq_size);
           if (!x) nrerror("Memory allocation error.\n");
      }
      else{
           x = a;
           while (x->next_freq_node != NULL) x = x->next_freq_node;
           y = farmalloc(freq_size);
           if (!y) nrerror("Memory allocation error.\n");
           x = x->next_freq_node = y;
      }
      x->diode_volts = farmalloc(number_of_diodes * sizeof(double));
      if (!(x->diode_volts)) nrerror("Memory allocation error.\n");
      x->next_freq_node = NULL;
      x->first_cal_node = NULL;
      x->center_r[1] = x->center_i[1] = 0.0;
      x->scale[1] = 1.0;
      return(x);
}
```

```
/***********************************************************************/
/*                                                                     */
int gaussj1(double a[7][7], int n, double b[7][2], int m)
/*                                                                     */
/* Gauss-Jordan matrix solver.  Matrix to be solved is in a, right-    */
/* hand vector is in b.  Details of operation are described in         */
/* Numerical Recipes                                                   */
/*                                                                     */
/***********************************************************************/
{
      double   big, big_one, dum, pivinv;
      int  i, icol, irow, j, k, l, ll;
      int  indxc[7], indxr[7], ipiv[7], singular;

      singular = 0;
      for (j = 1; j <= n; ++j){
           ipiv[j] = 0;
      }
      for (i = 1; i <= n; ++i){
           big = 0.0;
           for (j = 1; j <= n; ++j){
                if (ipiv[j] != 1){
                     for (k = 1; k <= n; ++k){
                          if (ipiv[k] == 0){
                               if (fabs(a[j][k]) >= big){
                                    big = fabs(a[j][k]);
                                    irow = j;
                                    icol = k;
                               }
                          }
                          else if (ipiv[k] > 1){
                               singular = 1;
                               goto quit;
                          }
                     }
                }
           }
           if (i == 1) big_one = big;
           ipiv[icol] = ipiv[icol]+1;
           if (irow != icol){
                for (l = 1; l <= n; ++l){
                     dum = a[irow][l];
                     a[irow][l] = a[icol][l];
                     a[icol][l] = dum;
```

```
                    }
                    for (l = 1; l <= m; ++l){
                         dum = b[irow][l];
                         b[irow][l] = b[icol][l];
                         b[icol][l] = dum;
                    }
               }
               indxr[i] = irow;
               indxc[i] = icol;
               if (fabs(a[icol][icol]) < ((1e-10)*big_one)){
                    singular = 1;
                    goto quit;
               }
               pivinv = 1.0/a[icol][icol];
               a[icol][icol] = 1.0;
               for (l = 1; l <= n; ++l){
                    a[icol][l] = a[icol][l]*pivinv;
               }
               for (l = 1; l <= m; ++l){
                    b[icol][l] = b[icol][l]*pivinv;
               }
               for (ll = 1; ll <= n; ++ll){
                    if (ll != icol){
                         dum = a[ll][icol];
                         a[ll][icol] = 0.0;
                         for (l = 1; l <= n; ++l){
                              a[ll][l] = a[ll][l]-a[icol][l]*dum;
                         }
                         for (l = 1; l <= m; ++l){
                              b[ll][l] = b[ll][l]-b[icol][l]*dum;
                         }
                    }
               }
          }
          for (l = n; l >= 1; --l){
               if (indxr[l] != indxc[l]){
                    for (k = 1; k <= n; ++k){
                         dum = a[k][indxr[l]];
                         a[k][indxr[l]] = a[k][indxc[l]];
                         a[k][indxc[l]] = dum;
                    }
               }
          }
quit: return(singular);
```

```
}

/*************************************************************************/
/*                                                                       */
void find_w(freq_node *a, double diode_volts[number_of_diodes], complex *w,
                                    int dsel)
/*                                                                       */
/* Finds complex ratio w using previously calculated calibration         */
/* constants and trigonometry.                                           */
/*                                                                       */
/*************************************************************************/
{
      int  i, j, error1;
      double   dum1, dum2, dum3, r1, r2, r3, cos_theta, sin_theta,
           x[2], err1, err2;

      dum1 = diode_volts[0];
      dum2 = diode_volts[1];
      dum3 = diode_volts[dsel];
      r1 = dum2/dum1;
      r2 = a->scale[dsel]*dum3/dum1;
      r3 = sqr(a->center_mag[dsel]);
      cos_theta = (r1 + r3 - r2)/(2*sqrt(r1)*sqrt(r3));
      if (fabs(cos_theta) > 1.0) cos_theta
           = cos_theta/fabs(cos_theta);
      sin_theta = -sqrt(1-sqr(cos_theta));
      *w = *co(sqrt(r1)*cos_theta, sqrt(r1)*sin_theta);
      reset_complex_pointer();
}
```

```
/***********************************************************************/
/*                                                                     */
void fill_complex_matrix(freq_node *a, double mat1[7][7],
                              double vect1[7][2])
/*                                                                     */
/* Fills matrix for 4-port (Short/Open/Load) calibration.  Note:       */
/* models for all calibration standards are contained in this          */
/* routine.  Edit here to change models.                               */
/*                                                                     */
/***********************************************************************/

{
      complex  gamma1, gamma2, gamma3, z1;
      double short_length, c1;

      mat1[1][3] = 1.0;
      mat1[3][3] = 1.0;
      mat1[5][3] = 1.0;
      mat1[2][4] = 1.0;
      mat1[4][4] = 1.0;
      mat1[6][4] = 1.0;
      mat1[2][3] = 0.0;
      mat1[4][3] = 0.0;
      mat1[6][3] = 0.0;
      mat1[1][4] = 0.0;
      mat1[3][4] = 0.0;
      mat1[5][4] = 0.0;

      /* First cal standard is a delayed short circuit.  Shorting plane */
      /* is recessed about 50 mils from reference plane.  Effective     */
      /* length given here was derived from model fitting using 8410    */
      /* measurements. */

      short_length = 1.1813e-3;
      gamma1 = *co(-cos(2*pi*(a->frequency)*1e9/v_1*2.0*short_length),
                    sin(2*pi*(a->frequency)*1e9/v_1*2.0*short_length));

      /* Second standard is a 7mm open circuit.  Its capacitance is    */
      /* modeled as c=0.079 + 4e-5*f^2, with f in GHz and c in pF.     */

      c1 = (0.079 + 4E-5 * (a->frequency) * (a->frequency)) * 1E-12;
      z1 = *co(0,-1.0 / (2.0 * pi * (a->frequency) * 1E9 * c1 * [illegible].0));
      gamma2 = *qu(di(&z1,co(1.0,0.0)),su(&z1,co(1.0,0.0)));
```

```
        /* Last standard is a matched load.  Model used is DC value of  */
        /* resistance shunted with a capacitor, and a series inductor   */
        /* between this shunt pair and the reference plane.  Model was  */
        /* derived by fitting to a large number of 8410 measurements.   */

        z1 = *qu(co(0.0, -49.64 / (2.0*pi*(a->frequency)*1e-3 * 8.9454e-3)),
                 co(49.64, -1.0 / (2.0*pi*(a->frequency)*1e-3 * 8.9454e-3)));
        z1 = *su(&z1, co(0.0, 2.0*pi*(a->frequency)* 1.9681e-3));
        gamma3 = *qu(di(&z1,co(50.0,0.0)),su(&z1,co(50.0,0.0)));
        gamma3 = *pr(&gamma3,
                 co(cos(2*pi*(a->frequency)*1e9/v_l*2.0 * 3.4564e-2),
                 -sin(2*pi*(a->frequency)*1e9/v_l*2.0 * 3.4564e-2)));

        /* Fill matrix for determination of 4-port calibration constants */
        /* as described in text. */

        mat1[1][5] = gamma1.x;
        mat1[2][6] = gamma1.x;
        mat1[2][5] = -gamma1.y;
        mat1[1][6] = gamma1.y;
        mat1[3][5] = gamma2.x;
        mat1[4][6] = gamma2.x;
        mat1[4][5] = -gamma2.y;
        mat1[3][6] = gamma2.y;
        mat1[5][5] = gamma3.x;
        mat1[6][6] = gamma3.x;
        mat1[6][5] = -gamma3.y;
        mat1[5][6] = gamma3.y;
        mat1[1][1] = -(gamma1.x*vect1[2][1] - gamma1.y*vect1[1][1]);
        mat1[2][2] = mat1[1][1];
        mat1[1][2] = -(gamma1.x*vect1[1][1] + gamma1.y*vect1[2][1]);
        mat1[2][1] = -mat1[1][2];
        mat1[3][1] = -(gamma2.x*vect1[4][1] - gamma2.y*vect1[3][1]);
        mat1[4][2] = mat1[3][1];
        mat1[3][2] = -(gamma2.x*vect1[3][1] + gamma2.y*vect1[4][1]);
        mat1[4][1] = -mat1[3][2];
        mat1[5][1] = -(gamma3.x*vect1[6][1] - gamma3.y*vect1[5][1]);
        mat1[6][2] = mat1[5][1];
        mat1[5][2] = -(gamma3.x*vect1[5][1] + gamma3.y*vect1[6][1]);
        mat1[6][1] = -mat1[5][2];

        reset_complex_pointer();
}
```

```
/*************************************************************************/
/*                                                                       */
void fill_complex_vector(freq_node *a, double vect1[7][2])
/*                                                                       */
/* Fills right-hand vector used in solving for 4-port calibration       */
/* constants.                                                            */
/*                                                                       */
/*************************************************************************/

{
      int  i, j;
      complex   w;

      find_w(a, a->z1_volts, &w, 2);
      vect1[2][1] = w.x;
      vect1[1][1] = w.y;
      find_w(a, a->z2_volts, &w, 2);
      vect1[4][1] = w.x;
      vect1[3][1] = w.y;
      find_w(a, a->z3_volts, &w, 2);
      vect1[6][1] = w.x;
      vect1[5][1] = w.y;
}
```

```
/**********************************************************************/
/*                                                                    */
void four_port_cal(freq_node *root)
/*                                                                    */
/* Solves for 4-port calibration constants.  Fills a complex matrix   */
/* with data from measurements of Short/Open/Load cal standards and   */
/* models of same.  Inverts matrix using Gauss-Jordan elimination     */
/* to yield calibration constants.                                    */
/*                                                                    */
/**********************************************************************/

{
      freq_node *a;
      double    (*vect1)[7][2], (*mat1)[7][7];
      int  i, j, k;
      char out_line[30];

      vect1 = farcalloc(14,sizeof(double));
      mat1 = farcalloc(49,sizeof(double));
      a = root;
      while (a != NULL){
           fill_complex_vector(a,*vect1);
           fill_complex_matrix(a,*mat1,*vect1);
           gaussj1(*mat1,6,*vect1,1);
           for (i = 1; i <= 6; ++i)
                a->bilin[0][i] = (*vect1)[i][1];
           a = a->next_freq_node;
      }
      farfree(mat1);
      farfree(vect1);
}
```

```
/*************************************************************************/
/*                                                                       */
void find_S(freq_node *a)
/*                                                                       */
/* Finds reflection coefficient.  First finds complex ratio w, then      */
/* performs bilinear transform to yield reflection coefficient as        */
/* described in text.                                                    */
/*                                                                       */
/*************************************************************************/

{
      double    temp1r, temp1i, temp2r, temp2i, temp3, (*mat1)[7][7],
           (*vect1)[7][2];
      int  i, j;
      complex   *rho[max_heads], a2_on_a1, temp_root1, w,
           temp_root2;

      find_w(a, a->meas_volts, &w, 2);
      temp1r = w.x - a->bilin[0][4];
      temp1i = w.y - a->bilin[0][3];
      temp2r = w.x * a->bilin[0][2]
           - w.y * a->bilin[0][1];
      temp2i = w.x * a->bilin[0][1]
           + w.y * a->bilin[0][2];
      temp2r = a->bilin[0][6] - temp2r;
      temp2i = a->bilin[0][5] - temp2i;
      temp3 = sqr(temp2r) + sqr(temp2i);
      a->S[0].x = (temp1r*temp2r + temp1i*temp2i)/temp3;
      a->S[0].y = (temp1i*temp2r - temp1r*temp2i)/temp3;
}
```

```
/*************************************************************************/
/*                                                                       */
int permute(double x[number_of_diodes], double frequency)
/*                                                                       */
/* Rearranges array x as a function of frequency such that the three    */
/* values to be used at a given frequency are in elements 0,1, and 2.   */
/*                                                                       */
/*************************************************************************/

{
      double temp;

      if ((frequency >= 0.0) && (frequency < 2)){
           temp = x[2];
           x[2] = x[4];
           x[4] = temp;
           return 0;
      }
      else if ((frequency >= 2) && (frequency <= 4)){
           temp = x[0];
           x[0] = x[1];
           x[1] = x[2];
           x[2] = x[4];
           x[4] = temp;
           return 0;
      }
      else if ((frequency > 4) && (frequency <= 8)){
           temp = x[0];
           x[0] = x[2];
           x[2] = x[4];
           x[4] = x[1];
           x[1] = x[3];
           x[3] = temp;
           return 0;
      }
      else return 1;
}
```

```
/**********************************************************************/
/*                                                                    */
double *vector(int nl, int nh)
/*                                                                    */
/* Allocates memory for a vector                                      */
/*                                                                    */
/**********************************************************************/

{
      double *v;

      v=(double *)malloc((unsigned) (nh-nl+1)*sizeof(double));
      if (!v) nrerror("allocation failure in vector()");
      return v-nl;
}

/**********************************************************************/
/*                                                                    */
int *ivector(int nl, int nh)
/*                                                                    */
/* Allocates memory for an integer vector.                            */
/*                                                                    */
/**********************************************************************/

{
      int *v;

      v=(int *)malloc((unsigned) (nh-nl+1)*sizeof(int));
      if (!v) nrerror("allocation failure in ivector()");
      return v-nl;
}
```

```
/************************************************************************/
/*                                                                      */
double **dmatrix(int nrl, int nrh, int ncl, int nch)
/*                                                                      */
/* Allocates memory for a 2-D array.                                    */
/*                                                                      */
/************************************************************************/

{
      int i;
      double **m;

      m=(double **) malloc((unsigned) (nrh-nrl+1)*sizeof(double*));
      if (!m) nrerror("allocation failure 1 in dmatrix()");
      m -= nrl;

      for(i=nrl;i<=nrh;i++) {
           m[i]=(double *) malloc((unsigned) (nch-ncl+1)*sizeof(double));
           if (!m[i]) nrerror("allocation failure 2 in dmatrix()");
           m[i] -= ncl;
      }
      return m;
}

/************************************************************************/
/*                                                                      */
void free_ivector(v,nl,nh)
int *v,nl,nh;
/*                                                                      */
/* Frees memory allocated for an integer vector.                        */
/*                                                                      */
/************************************************************************/

{
      free((char*) (v+nl));
}
```

```
/***********************************************************************/
/*                                                                     */
void free_dmatrix(double **m, int nrl, int nrh, int ncl, int nch)
/*                                                                     */
/* Frees memory allocated for a 2-D array.                             */
/*                                                                     */
/***********************************************************************/

{
      int i;

      for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
      free((char*) (m+nrl));
}

/***********************************************************************/
/*                                                                     */
void free_vector(double *v, int nl, int nh)
/*                                                                     */
/* Frees memory allocated for a vector.                                */
/*                                                                     */
/***********************************************************************/

{
      free((char*) (v+nl));
}
```

```
/***********************************************************************/
/*                                                                     */
void svbksb(double **u, double w[], double **v, int m, int n,
                              double b[], double x[])
/*                                                                     */
/* Performs back substitution for singular value decomposition.       */
/*                                                                     */
/***********************************************************************/

{
      int jj,j,i;
      double s,*tmp;

      tmp=vector(1,n);
      for (j=1;j<=n;j++) {
           s=0.0;
           if (w[j]) {
                for (i=1;i<=m;i++) s += u[i][j]*b[i];
                s /= w[j];
           }
           tmp[j]=s;
      }
      for (j=1;j<=n;j++) {
           s=0.0;
           for (jj=1;jj<=n;jj++) s += v[j][jj]*tmp[jj];
           x[j]=s;
      }
      free_vector(tmp,1,n);
}
```

```
/***********************************************************************/
/*                                                                     */
void svdcmp(double **a, int m, int n, double *w, double **v)
/*                                                                     */
/* Performs singular value decomposition of matrix a.  Robust tech-   */
/* nique for matrix inversion which works well in parameter esti-     */
/* mation applications.  See numerical recipes for details.           */
/*                                                                     */
/***********************************************************************/

{
      int flag,i,its,j,jj,k,l,nm;
      double c,f,h,s,x,y,z;
      double anorm=0.0,g=0.0,scale=0.0;
      double *rv1;

      if (m < n) nrerror("SVDCMP: You must augment A with extra zero rows");
      rv1=vector(1,n);
      for (i=1;i<=n;i++) {
           l=i+1;
           rv1[i]=scale*g;
           g=s=scale=0.0;
           if (i <= m) {
                for (k=i;k<=m;k++) scale += fabs(a[k][i]);
                if (scale) {
                     for (k=i;k<=m;k++) {
                          a[k][i] /= scale;
                          s += a[k][i]*a[k][i];
                     }
                     f=a[i][i];
                     g = -SIGN(sqrt(s),f);
                     h=f*g-s;
                     a[i][i]=f-g;
                     if (i != n) {
                          for (j=l;j<=n;j++) {
                               for (s=0.0,k=i;k<=m;k++) s += a[k][i]*a[k][j];
                               f=s/h;
                               for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
                          }
                     }
                     for (k=i;k<=m;k++) a[k][i] *= scale;
                }
           }
           w[i]=scale*g;
```

```
            g=s=scale=0.0;
            if (i <= m && i != n) {
                 for (k=l;k<=n;k++) scale += fabs(a[i][k]);
                 if (scale) {
                      for (k=l;k<=n;k++) {
                           a[i][k] /= scale;
                           s += a[i][k]*a[i][k];
                      }
                      f=a[i][l];
                      g = -SIGN(sqrt(s),f);
                      h=f*g-s;
                      a[i][l]=f-g;
                      for (k=l;k<=n;k++) rv1[k]=a[i][k]/h;
                      if (i != m) {
                           for (j=l;j<=m;j++) {
                                for (s=0.0,k=l;k<=n;k++) s += a[j][k]*a[i][k];
                                for (k=l;k<=n;k++) a[j][k] += s*rv1[k];
                           }
                      }
                      for (k=l;k<=n;k++) a[i][k] *= scale;
                 }
            }
            anorm=MAX(anorm,(fabs(w[i])+fabs(rv1[i])));
       }
       for (i=n;i>=1;i--) {
            if (i < n) {
                 if (g) {
                      for (j=l;j<=n;j++)
                           v[j][i]=(a[i][j]/a[i][l])/g;
                      for (j=l;j<=n;j++) {
                           for (s=0.0,k=l;k<=n;k++) s += a[i][k]*v[k][j];
                           for (k=l;k<=n;k++) v[k][j] += s*v[k][i];
                      }
                 }
                 for (j=l;j<=n;j++) v[i][j]=v[j][i]=0.0;
            }
            v[i][i]=1.0;
            g=rv1[i];
            l=i;
       }
       for (i=n;i>=1;i--) {
            l=i+1;
            g=w[i];
            if (i < n)
```

```
                for (j=1;j<=n;j++) a[i][j]=0.0;
            if (g) {
                g=1.0/g;
                if (i != n) {
                    for (j=1;j<=n;j++) {
                        for (s=0.0,k=1;k<=m;k++) s += a[k][i]*a[k][j];
                        f=(s/a[i][i])*g;
                        for (k=i;k<=m;k++) a[k][j] += f*a[k][i];
                    }
                }
                for (j=i;j<=m;j++) a[j][i] *= g;
            } else {
                for (j=i;j<=m;j++) a[j][i]=0.0;
            }
            ++a[i][i];
        }
        for (k=n;k>=1;k--) {
            for (its=1;its<=30;its++) {
                flag=1;
                for (l=k;l>=1;l--) {
                    nm=l-1;
                    if (fabs(rv1[l])+anorm == anorm) {
                        flag=0;
                        break;
                    }
                    if (fabs(w[nm])+anorm == anorm) break;
                }
                if (flag) {
                    c=0.0;
                    s=1.0;
                    for (i=l;i<=k;i++) {
                        f=s*rv1[i];
                        if (fabs(f)+anorm != anorm) {
                            g=w[i];
                            h=PYTHAG(f,g);
                            w[i]=h;
                            h=1.0/h;
                            c=g*h;
                            s=(-f*h);
                            for (j=1;j<=m;j++) {
                                y=a[j][nm];
                                z=a[j][i];
                                a[j][nm]=y*c+z*s;
                                a[j][i]=z*c-y*s;
                        }
                    }
                }
            }
            z=w[k];
            if (l == k) {
                if (z < 0.0) {
                    w[k] = -z;
                    for (j=1;j<=n;j++) v[j][k]=(-v[j][k]);
                }
                break;
            }
            if (its == 30) nrerror("No convergence in 30 SVDCMP iterations");
            x=w[l];
            nm=k-1;
            y=w[nm];
            g=rv1[nm];
            h=rv1[k];
            f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0*h*y);
            g=PYTHAG(f,1.0);
            f=((x-z)*(x+z)+h*((y/(f+SIGN(g,f)))-h))/x;
            c=s=1.0;
            for (j=l;j<=nm;j++) {
                i=j+1;
                g=rv1[i];
                y=w[i];
                h=s*g;
                g=c*g;
                z=PYTHAG(f,h);
                rv1[j]=z;
                c=f/z;
                s=h/z;
                f=x*c+g*s;
                g=g*c-x*s;
                h=y*s;
                y=y*c;
                for (jj=1;jj<=n;jj++) {
                    x=v[jj][j];
                    z=v[jj][i];
                    v[jj][j]=x*c+z*s;
                    v[jj][i]=z*c-x*s;
                }
                z=PYTHAG(f,h);
                w[j]=z;
```

```
                    if (z) {
                         z=1.0/z;
                         c=f*z;
                         s=h*z;
                    }
                    f=(c*g)+(s*y);
                    x=(c*y)-(s*g);
                    for (jj=1;jj<=m;jj++) {
                         y=a[jj][j];
                         z=a[jj][i];
                         a[jj][j]=y*c+z*s;
                         a[jj][i]=z*c-y*s;
                    }
               }
               rv1[l]=0.0;
               rv1[k]=f;
               w[k]=x;
          }
     }
     free_vector(rv1,1,n);
}

/************************************************************************/
/*                                                                      */
void wait_microsec(int arg1)
/*                                                                      */
/* Assembly language timing loop for short delays.  DOS and BIOS        */
/*                                                                      */
/************************************************************************/

{
asm   mov  cx,arg1
s1:
asm   nop
asm   loop s1
}

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

void interrupt (*oldfunc)();
```

```
/***********************************************************************/
/*                                                                     */
void mode_set(char mode)
/*                                                                     */
/* Sets mode of programmable peripheral interface on PIO-12 board.    */
/*                                                                     */
/***********************************************************************/

{
      outportb(pio12_control, mode);
}
```

```
/*************************************************************************/
/*                                                                       */
void dt_error_check()
/*                                                                       */
/* This routine is called whenever an error is reported by the Data      */
/* Translation board.  It queries the board, interprets the resulting */
/* error code, and prints an error message.                              */
/*                                                                       */
/*************************************************************************/

{
      int  i;
      int  ready = 0;
      int  timeout = -32767;
      char a, b, reg;

      for (i = 0; i < 100; ++i){}
      outportb(dt_2801_command_register, READ_ERR);
      while (ready == 0){
           for (i = 0; i < 100; ++i){}
           reg = inportb(dt_2801_status_register);
           if ((reg & DOR) == DOR)
                ready = 1;
           else{
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
```

```
                    printf("during dt_error_check.\n");
                    ready = 1;
               }
          }
     }
     a = inportb(dt_2801_data_register);
     ready = 0;
     timeout = -32767;
     while (ready == 0){
          for (i = 0; i < 100; ++i){}
          reg = inportb(dt_2801_status_register);
          if ((reg & DOR) == DOR)
               ready = 1;
          else{
               ++timeout;
               if (timeout == 32767){
                    printf("Device time-out on DT-2801 board,\n");
                    printf("during dt_error_check.\n");
                    ready = 1;
               }
          }
     }
     b = inportb(dt_2801_data_register);
     if ((a & 0x02) == 0x02)
          printf("Command Overwrite Error\n");
     if ((a & 0x04) == 0x04)
          printf("Clock Set Error\n");
     if ((a & 0x08) == 0x08)
          printf("Digital Port Select Error\n");
     if ((a & 0x10) == 0x10)
          printf("Digital Port Set Error\n");
     if ((a & 0x20) == 0x20)
          printf("DAC Select Error\n");
     if ((a & 0x40) == 0x40)
          printf("DAC Clock Error\n");
     if ((a & 0x80) == 0x80)
          printf("DAC No. Conversions Value Error\n");
     if ((b & 0x01) == 0x01)
          printf("A/D Channel Error\n");
     if ((b & 0x02) == 0x02)
          printf("A/D Gain Error\n");
     if ((b & 0x04) == 0x04)
          printf("A/D Clock Error\n");
     if ((b & 0x08) == 0x08)
          printf("A/D Multiplexer Error\n");
     if ((b & 0x10) == 0x10)
          printf("A/D No. Conversions Value Error\n");
     if ((b & 0x20) == 0x20)
          printf("Data Where Command Expected Error\n");
}
```

```
/*************************************************************************/
/*                                                                       */
void dt_set_wait(unsigned char bytecode)
/*                                                                       */
/* Part of the handshaking with the Data Translation board, this         */
/* routine waits until a particular byte pattern is set in the DT-       */
/* 2801's status register.                                               */
/*                                                                       */
/*************************************************************************/

{
      char ready, i, idle;
      int timeout;

      ready = 0;
      timeout = -32767;
      while (ready == 0){
           idle = 0;
           while (idle <= 100) ++idle;
           i = inportb(dt_2801_status_register);
           if ((i & 0x80) == 0x80) dt_error_check();
           if ((i & bytecode) == bytecode)
                ready = 1;
           else
           {
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
                     printf("during dt_set_wait.\n");
                     ready = 1;
                }
           }
      }
}
```

```
/***********************************************************************/
/*                                                                     */
void dt_set_wait_no_error(unsigned char bytecode)
/*                                                                     */
/* Same as dt_set_wait except this routine does not check for errors  */
/* from the DT-2801 board                                              */
/*                                                                     */
/***********************************************************************/

{
      char ready = 0, i, idle;
      int timeout;

      timeout = -32767;
      while (ready == 0){
           idle = 0;
           while (idle <= 100) ++idle;
           i = inportb(dt_2801_status_register);
           if ((i & bytecode) == bytecode)
                ready = 1;
           else
           {
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
                     printf("during dt_set_wait.\n");
                     ready = 1;
                }
           }
      }
}
```

```
/************************************************************************/
/*                                                                      */
void dt_clear_wait(unsigned char bytecode)
/*                                                                      */
/* Part of the handshaking with the Data Translation board, this       */
/* routine waits until a particular byte pattern is cleared in the     */
/* DT-2801's status register.                                           */
/*                                                                      */
/************************************************************************/

{
      char ready, i, idle;
      int timeout;

      ready = 0;
      timeout = -32767;
      while (ready == 0){
           idle = 0;
           while (idle <= 100) ++idle;
           i = inportb(dt_2801_status_register);
           if ((i & 0x80) == 0x80) dt_error_check();
           if ((i & bytecode) == 0)
                ready = 1;
           else
           {
                ++timeout;
                if (timeout == 32767){
                     printf("Device time-out on DT-2801 board,\n");
                     printf("during dt_clear_wait.\n");
                     ready = 1;
                }
           }
      }
}
```

```
/***********************************************************************/
/*                                                                     */
void dt_2801_init()
/*                                                                     */
/* This routine initializes the Data Translation board.  It first     */
/* resets the board, then reads its identity and prints out the board */
/* type.  Then it executes the boards self-test, and sets both        */
/* digital ports for output.                                           */
/*                                                                     */
/***********************************************************************/

{
      unsigned char status, test_val, id, revno;
      int  i;

      status = inportb(dt_2801_status_register);
      if ((status & 0x70) != 0){
           printf("Illegal status register value\n");
           exit(1);
      }
      outportb(dt_2801_command_register, STOP); /* Stops execution of any */
      for (i = 0; i < 100; ++i){}               /* existing commands.     */
      status = inportb(dt_2801_data_register);  /* Dummy read to clear    */
      dt_set_wait_no_error(READY);              /* registers.             */
      outportb(dt_2801_command_register, RESET);/* Reset board.           */
      dt_set_wait_no_error(DOR);
      status = inportb(dt_2801_data_register);  /* Read board ID          */
      dt_set_wait(READY);
      id = status & 0xF0;
      revno = status & 0x0F;
                                         /* Decode board ID number */
/*    switch(id){
           case 0x00:    printf("DT-2801 board,");
                    break;
           case 0x10:    printf("DT-2805 board,");
                    break;
           case 0x20:    printf("DT-2808 board,");
                    break;
           case 0x30:    printf("DT-2808 board, with extender,");
                    break;
           case 0x50:    printf("DT-2801-A board,");
                    break;
           case 0x80:    printf("DT-2801/5716 board,");
                    break;
```

```
            case 0x90:    printf("DT-2805/5716 board,");
                          break;
            case 0xA0:    printf("DT-2818 board,");
        }
        printf(" firmware revision %2d.\n", revno);*/
        outportb(dt_2801_command_register, TEST);     /* Self test routine.  */
        for (i = 1; i < 256; ++i){                    /* Board should output */
            dt_set_wait(DOR);                       /* sequential values.  */
            test_val = inportb(dt_2801_data_register);
            status = inportb(dt_2801_status_register);
            if ((test_val != i) || ((status & 0x80) != 0)){
                printf("DT-2801 Failure in TEST Routine\n");
                exit(1);
            }
        }
        dt_set_wait(DOR);
        test_val = inportb(dt_2801_data_register);
        status = inportb(dt_2801_status_register);
        if ((test_val != 0) || ((status & 0x80) != 0)){
            printf("DT-2801 Failure in TEST Routine\n");
            exit(1);
        }
        outportb(dt_2801_command_register, STOP);      /* Stop self-test.    */
        for (i = 0; i < 100; ++i){}
        status = inportb(dt_2801_data_register);
        dt_set_wait_no_error(READY);
        outportb(dt_2801_command_register, RESET);    /* Reset board.        */
        dt_set_wait_no_error(DOR);
        status = inportb(dt_2801_data_register);
        dt_set_wait(READY);
        outportb(dt_2801_command_register,SET_DIGITAL_OUTPUT); /* Set up     */
        dt_clear_wait(DIF);                         /* both digital ports for  */
        outportb(dt_2801_data_register,BOTH);       /* output.                 */
        dt_clear_wait(DIF);
        dt_set_wait(READY);
}
```

```
/**************************************************************************/
/*                                                                        */
void write_digital(unsigned char byte_select, unsigned char out_byte)
/*                                                                        */
/* This I/O routine writes the byte out_byte to the DT-2801 parallel     */
/* port selected by byte_select.  The value of byte_select can be 1      */
/* or 0 since the DT-2801 has only 2 ports.                               */
/*                                                                        */
/**************************************************************************/

{
      int  i;

      outportb(dt_2801_command_register, WRITE_DIG_IMM);
      dt_clear_wait(DIF);
      outportb(dt_2801_data_register, byte_select);
      dt_clear_wait(DIF);
      outportb(dt_2801_data_register, out_byte);
      dt_clear_wait(DIF);
      dt_set_wait(READY);
}
```

```
/*************************************************************************/
/*                                                                       */
void write_to_YTF(unsigned int value)
/*                                                                       */
/* This I/O routine writes the integer value to the YIG-tuned            */
/* filter's control port.                                                */
/*                                                                       */
/*************************************************************************/

{
      unsigned char a;

      value += YTFTWEAK;
      YTFLSB = value;
      write_digital(0, ~YTFLSB);
      PORT1 = PORT1 & 0xF8;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
      UPPERS = ((value >> 8) & 0x0F) | (UPPERS & 0xF0);
      write_digital(0, ~UPPERS);
      PORT1 = (PORT1 & 0xF8) | 0x02;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
}
```

```
/************************************************************************/
/*                                                                      */
void write_to_YTO(unsigned int value)
/*                                                                      */
/* This I/O routine writes the integer value to the YIG-tuned           */
/* oscillator's control port.                                           */
/*                                                                      */
/************************************************************************/

{
      YTOLSB = value;
      write_digital(0, ~YTOLSB);
      PORT1 = (PORT1 & 0xF8) | 0x01;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
      UPPERS = ((value >> 4) & 0xF0) | (UPPERS & 0x0F);
      write_digital(0, ~UPPERS);
      PORT1 = (PORT1 & 0xF8) | 0x02;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
}
```

```
/************************************************************************/
/*                                                                      */
void write_to_VCO(unsigned int value)
/*                                                                      */
/* Similar to write_to_YTF, this routine programs the signal           */
/* generator's voltage-controlled oscillator.                          */
/*                                                                      */
/************************************************************************/

{
      VCOMSB = (value >> 4) & 0xF0;
      write_digital(0, ~VCOMSB);
      PORT1 = (PORT1 & 0xF8) | 0x05;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
      VCOLSB = value;
      write_digital(0, ~VCOLSB);
      PORT1 = (PORT1 & 0xF8) | 0x04;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
}
```

```
/*************************************************************************/
/*                                                                       */
void write_switch(unsigned char value)
/*                                                                       */
/* Similar to write_to_YTF, this routine programs the signal             */
/* generator's internal selector switch.                                 */
/*                                                                       */
/*************************************************************************/

{
      CONTROL = value;
      write_digital(0, ~CONTROL);
      PORT1 = (PORT1 & 0xF8) | 0x03;
      write_digital(1, PORT1);
      PORT1 = PORT1 & 0xF7;
      write_digital(1, PORT1);
      PORT1 = (PORT1 | 0x08);
      write_digital(1, PORT1);
}
```

```
/*************************************************************************/
/*                                                                       */
void to_frequency(double frequency)
/*                                                                       */
/* This routine uses write_to_YTF, write_to_YTO, write_to_VCO, and       */
/* write_switch to command the signal generator to the specified         */
/* frequency.  To keep the filter aligned with the oscillator            */
/* frequencies, the global array cal_store[] is referenced.              */
/* cal_store[] contains a look-up table generated from an alignment      */
/* routine which gives the oscillator code which lines up with a         */
/* given filter code value.                                              */
/*                                                                       */
/*************************************************************************/

{
      int  i;
      unsigned int YTF_code, YTO_code, VCO_code;
      unsigned char CONTROL_code;
      double    temp, temp1;
      static double YTF_freq[6] = {1.0, 2.691, 4.394, 6.098, 7.798, 8.0};
      static double YTF_val[6] = {0.0, 994.0, 1989.0, 2983.0, 3977.0,
           4095.0};

      if ((frequency <= 8.0) && (frequency >= 2.0)){
           CONTROL_code = 0x02;
           write_switch(CONTROL_code);
           for (i = 0; i < 6; ++i){
                temp = frequency - YTF_freq[i];
                if (temp <= 0) break;
           }
           temp = YTF_val[i-1] + (frequency - YTF_freq[i-1])
                     / (YTF_freq[i] - YTF_freq[i-1])
                          * (YTF_val[i] - YTF_val[i-1]);
           temp1 = modf(temp, &temp);
           if (temp1 > 0.5) temp = temp + 1.0;
           YTF_code = (unsigned int) temp;
           write_to_YTF(YTF_code);
           write_to_YTO(calstore[YTF_code]);
      }
      else if ((frequency < 2.0) && (frequency >= 1.0)){
           CONTROL_code = 0x00;
           write_switch(CONTROL_code);
           temp = YTF_val[0] + (frequency - YTF_freq[0])
                     / (YTF_freq[1] - YTF_freq[0])
                          * (YTF_val[1] - YTF_val[0]);
                temp1 = modf(temp, &temp);
                if (temp1 > 0.5) temp = temp + 1.0;
                YTF_code = (unsigned int) temp;
                write_to_YTF(YTF_code);
                write_to_VCO(calstore[YTF_code]);
        }
}
```

```
/***********************************************************************/
/*                                                                     */
void set_gains_int(char g1, char g2, char int1, char bias)
/*                                                                     */
/* This routine sets the gains of the various programmable ampli-     */
/* fiers in the network analyzer's post-detection electronics.  It    */
/* also sets the integration time and the bias current.               */
/* Communication with the post-detection electronics is through       */
/* the PIO-12 24-bit parallel interface card.                         */
/*                                                                     */
/***********************************************************************/

{
        outportb(pio12_pb, 0x14);
asm     jmp  $+2
        outportb(pio12_pa,
             ~((0x01 << (g1-1)) | (0x08 << (g2-1)) |
                                  ((bias << 7) & 0x80)));
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2
        outportb(pio12_pc, 0x80);
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2

        outportb(pio12_pb, 0x15);
asm     jmp  $+2
        outportb(pio12_pa, ~(0x01 << (int1-1)));
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2
        outportb(pio12_pc, 0x80);
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2
}
```

```
/*************************************************************************/
/*                                                                       */
void data_in()
/*                                                                       */
/* Set PIO-12 board up for input on one of its 8-bit ports.  Keep        */
/* other 2 ports in output mode for control.                             */
/*                                                                       */
/*************************************************************************/

{
        char mode;

        outportb(pio12_pa, 0x00);
asm     jmp  $+2
        outportb(pio12_pb, 0x00);
asm     jmp  $+2
        outportb(pio12_pc, 0x00);
asm     jmp  $+2

        mode = pa_is_input | pb_is_output | pclo_is_output
             | pchi_is_output | pa_pchi_is_mode_0 | pb_pclo_is_mode_0
                  | mode_set_active;
        mode_set(mode);
}

/*************************************************************************/
/*                                                                       */
void data_out()
/*                                                                       */
/* Set PIO-12 board up for output on all of its 8-bit ports.             */
/*                                                                       */
/*************************************************************************/

{
        char mode;

        mode = pa_is_output | pb_is_output | pclo_is_output
             | pchi_is_output | pa_pchi_is_mode_0 | pb_pclo_is_mode_0
                  | mode_set_active;
        mode_set(mode);
}
```

```
/************************************************************************/
/*                                                                      */
void read_ads(char s_channel, char e_channel, unsigned int *out_vector)
/*                                                                      */
/* Uses the 16-bit A/D converter in the post-detection electronics     */
/* to read the analog outputs of the various synchronous detectors.    */
/* s_channel is first channel to be read, e_channel is last.  Array    */
/* of converted values is returned in out_vector.                       */
/*                                                                      */
/************************************************************************/

{
        char byte1[2], i;
        unsigned int   *temp;

        temp = (unsigned int *) byte1;

        outportb(pio12_pc, 0xC0);
        delay(2);
        outportb(pio12_pc, 0xC8);
        wait_microsec(100);
        for (i = s_channel; i <= e_channel; ++i){
              outportb(pio12_pb, 0x08 | (i & 0x07));
asm     jmp   $+2;
              outportb(pio12_pc, 0x88);
asm     jmp   $+2;
              outportb(pio12_pc, 0xC8);
              wait_microsec(10);
              outportb(pio12_pc, 0xE8);
asm     jmp   $+2;
              outportb(pio12_pc, 0xC8);
              wait_microsec(100);
              outportb(pio12_pb, 0x18);
asm     jmp   $+2;
              byte1[0] = inportb(pio12_pa);
asm     jmp   $+2;
              outportb(pio12_pb, 0x19);
asm     jmp   $+2;
              byte1[1] = inportb(pio12_pa);
              out_vector[i-s_channel] = ~(*temp);
        }
        outportb(pio12_pc, 0xC0);
}

/************************************************************************/
```

```
#include <conio.h>
#include <alloc.h>
#include <dos.h>
#include <fcntl.h>
#include <graphics.h>
#include <io.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys\stat.h>
#include <time.h>
#define sqr(x) ((x)*(x))
#define pio12_pa 0x200
#define pio12_pb 0x201
#define pio12_pc 0x202
#define pio12_control 0x203
#define pclo_is_input 0x01
#define pclo_is_output 0x00
#define pb_is_input 0x02
#define pb_is_output 0x00
#define pb_pclo_is_mode_0 0x00
#define pb_pclo_is_mode_1 0x04
#define pchi_is_input 0x08
#define pchi_is_output 0x00
#define pa_is_input 0x10
#define pa_is_output 0x00
#define pa_pchi_is_mode_0 0x00
#define pa_pchi_is_mode_1 0x20
#define pa_pchi_is_mode_2 0x40
#define set_reset_mode 0x00
#define mode_set_active 0x80
#define pi 3.14159265358979324
#define twopi 6.28318530717958648
#define v_l 299792458.0
#define   CW 0
#define SWEEP 1
#define CW_PULSE 2
#define SWEEP_PULSE 3
#define dt_2801_status_register 0x02ED
#define dt_2801_command_register 0x02ED
#define dt_2801_data_register 0x02EC
#define STOP 0x0F
#define RESET 0x00
```

```
#define CLEAR_ERROR 0x01
#define SET_INT_CLK 0x03
#define SET_DIGITAL_OUTPUT 0x05
#define TEST 0x0B
#define SET_A_D_PARAMS 0x0D
#define WRITE_DIG_IMM 0x07
#define READ_ERR 0x02
#define BOTH 0x02
#define DOR 0x01
#define DIF 0x02
#define READY 0x04
#define COMMAND 0x08
#define CERROR  0x80
#define Nslides 20
#define Ndiodes 5
#define max_heads 2
#define number_of_diodes 5
#define number_of_phases 12
#define number_of_terms 3
#define number_of_delays 5
#define pi 3.14159265358979324
#define twopi 6.28318530717958648
#define v_l 299792458.0
#define number_of_tests 10000
#define freq_size sizeof(freq_node)
#define cal_size sizeof(cal_node)
#define next_complex_value() ++complex_pointer
#define next_c_mat_value() ++c_mat_pointer
#define reset_complex_pointer() complex_pointer=complex_base
#define reset_c_mat_pointer() c_mat_pointer=c_mat_base
#define free_complex_mem() farfree(++complex_base);farfree(++c_mat_base)

typedef struct COMPLEX {double x,y;} complex;

typedef struct COMPLEX_MAT {complex a11, a12, a21, a22;} complex_mat;

typedef double *diode_array;

typedef struct cal_node1{
      double   diode_volts[number_of_diodes];
      struct cal_node1 *next_cal_node;
} cal_node;

typedef struct freq_node1{
        double    frequency, power_level, R_c[2], R[2];
        double    *diode_volts;
        double  z1_volts[number_of_diodes];
        double  z2_volts[number_of_diodes];
        double  z3_volts[number_of_diodes];
        double    meas_volts[number_of_diodes];
        double    center_mag[number_of_diodes];
        double    theta[number_of_diodes];
        double    center_r[number_of_diodes];
        double    center_i[number_of_diodes];
        double    scale[number_of_diodes];
        complex   S[max_heads];
        complex C[number_of_phases][3];
        double    bilin[2][7];
        struct    freq_node1    *next_freq_node;
        cal_node *first_cal_node;
} freq_node;

typedef struct NBS_node1{
        double    d_volts[4];
        double    center_mag[4];
        double    theta[4];
        double    center_r1[4];
        double    center_i1[4];
        double    scale1[4];
        complex   bilin[3];
} NBS_node;

static double at,bt,ct;
#define PYTHAG(a,b) ((at=fabs(a)) > (bt=fabs(b)) ? \
(ct=bt/at,at*sqrt(1.0+ct*ct)) : (bt ? (ct=at/bt,bt*sqrt(1.0+ct*ct)): 0.0))

static double maxarg1,maxarg2;
#define MAX(a,b) (maxarg1=(a),maxarg2=(b),(maxarg1) > (maxarg2) ?\
        (maxarg1) : (maxarg2))
#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))
#define TOL 1.0e-5

complex    *complex_base = NULL;
complex    *complex_pointer = NULL;
complex_mat *c_mat_base = NULL;
complex_mat *c_mat_pointer = NULL;

#define Nslides 20
```

```
#define Ndiodes 5

struct sweeper_status {
     int     mode;
     double      frequency, freq_step, start_freq, stop_freq, sweep_time;
};

unsigned char YTOLSB, YTFLSB, UPPERS, CONTROL, PORT1, VCOLSB, VCOMSB;
int    YTFTWEAK = 0, MOD_ON = 0, ODD_NUM = 0;
double     FOFFSET = 0.0, DELTAF = 0.0;
int    *calstore;

void nrerror(char error_text[]);

complex Complex(double re, double im);

void get_complex_mem(void);

complex *co(double x, double y);

complex *su(complex *a, complex *b);

complex *di(complex *a, complex *b);

complex *pr(complex *a, complex *b);

complex *qu(complex *a, complex *b);

double c_abs(complex *a);

complex *c_sqrt(complex *a);

complex    *rc_mul(double x, complex *a);

complex *c_exp(complex *z);

cal_node *add_cal_node(freq_node *a);

freq_node *add_freq_node(freq_node *a);

int gaussj1(double a[7][7], int n, double b[7][2], int m);

void find_w(freq_node *a, double diode_volts[number_of_diodes], complex *w,
                                        int dsel);
int sum_into_matrix(freq_node *a, double **mat1,
                double **vect1, int diodes[3]);

int sum_into_matrix1(freq_node *a, double **mat1, double **vect1);

void fill_complex_matrix(freq_node *a, double mat1[7][7],
                              double vect1[7][2]);

void fill_complex_vector(freq_node *a, double vect1[7][2]);

void four_port_cal(freq_node *root);

void find_S(freq_node *a);

int find_centers(double vect1[], double *center_mag, double *scale);

int permute(double x[number_of_diodes], double frequency);

int c_permute(complex x[number_of_diodes], double frequency);

double *vector(int nl, int nh);

int *ivector(int nl, int nh);

double **dmatrix(int nrl, int nrh, int ncl, int nch);

void free_ivector(int *v, int nl, int nh);

void free_dmatrix(double **m, int nrl, int nrh, int ncl, int nch);

void free_vector(double *v, int nl, int nh);

void gaussj(double **a, int n, double **b, int m);

void svbksb(double **u, double w[], double **v, int m, int n,
                              double b[], double x[]);

void svdcmp(double **a, int m, int n, double *w, double **v);

void svdfit(double x1[], double x2[], double y[], double sig[],
      int ndata, double a[], int ma, double **u, double **v, double w[],
           double *chisq, void (*funcs)(double, double, double *, int));
```

```
void wait_microsec(int arg1);

void interrupt (*oldfunc)();;

void bios_wait(unsigned int microseconds);

void mode_set(char mode);

void dt_error_check();

void dt_set_wait(unsigned char bytecode);

void dt_set_wait_no_error(unsigned char bytecode);

void dt_clear_wait(unsigned char bytecode);

void dt_2801_init();

void write_digital(unsigned char byte_select, unsigned char out_byte);

void write_to_YTF(unsigned int value);

void write_to_YTO(unsigned int value);

void write_to_VCO(unsigned int value);

void write_switch(unsigned char value);

void to_frequency(double frequency);

void set_gains_int(char g1, char g2, char int1, char bias);

void set_analog_channel(char c1);

void data_in();

void data_out();

void read_ads(char s_channel, char e_channel, unsigned int *out_vector);
```