https://ntrs.nasa.gov/search.jsp?R=19840018892 2020-03-20T21:59:07+00:00Z

N84-26960

Progress Report RSC-4242-4-2

DEVELOPMENT OF LAND BASED RADAR POLARIMETER PROCESSOR SYSTEM

by

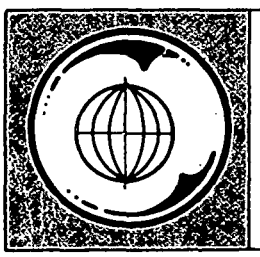
C. W. Kronke

A. J. Blanchard

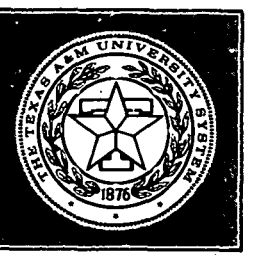
September 1983

Supported by

National Aeronautics and Space Administration
Goddard Space Flight Center
Greenbelt, Maryland 20771



TEXAS A&M UNIVERSITY
REMOTE SENSING CENTER
COLLEGE STATION, TEXAS



DEVELOPMENT OF LAND BASED RADAR POLARIMETER PROCESSOR SYSTEM

by

C. Kronke

A. J. Blanchard

September 1983

Supported by

National Aeronautics and Space Administration
Goddard Space Flight Center
Greenbelt, Maryland 20771

Contract No. NAG 5-31

ABSTRACT

The processing subsystem of a land based radar polarimeter was designed and constructed. This subsystem is labeled the Remote Data Acquisition and Distribution System (RDADS). The radar polarimeter, an experimental remote sensor, incorporates the RDADS to control all operations of the sensor. The RDADS uses Original Equipment Manufacturers (OEM) industrial standard components including an 8-bit microprocessor based single board computer, analog input/output (I/O) boards, a dynamic Random Access Memory (RAM) board, and power supplies. A high-speed digital electronics board was specially designed and constructed to control range-gating for the radar. A complete system of software programs was developed to operate the RDADS. The software uses a powerful real-time, multi-tasking, executive package as an operating system. The hardware and software used in the RDADS is presented here in detail. Also, recommendations for future system improvements are made.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

PROJECT BACKGROUND

The Remote Sensing Center (RSC) at Texas A&M University was contracted by NASA Goddard Space Flight Center to design, build, and put into operation a mobile land based radar polarimeter [1]. The formal name of this experimental remote sensor is the Radar Polarimeter System (RPS). The sensor was constructed for use in soil moisture and other agricultural remote sensing experiments. An integral component of the RPS is an electronic control and data acquisition system. This major RPS subsystem receives commands from and sends information to a master processing computer. The control and data acquisition electronics will hereafter be referred to as the Remote Data Acquisition and Distribution System or RDADS. It was the objective of this effort to design and construct the RDADS.

Remote Sensing

Remote Sensing is a science dealing with obtaining information about objects through measurements made without coming into contact with the objects. Currently this science is providing valuable information to those involved in agriculture, forestry, hydrology, mineral exploration, and land use management. For many years theorists and scientists have been interested in the depolarization effects of

natural terrain on electromagnetic energy (Blanchard [2], Fung [3] Leader [4], Rouse [5], Beckman [6], Blanchard [7], etc.). The phenomenon of the depolarization of radar backscatter has been established as having potential application in the remote sensing of soil moisture content (Blanchard [8], Hirosawa [9], Ulaby [10], etc.). There is, however, a need for controlled experimentation investigating the effect of target parameters on cross polarized radar measurements. The ground based radar polarimeter was constructed for use in this type of experimentation.

The Radar Polarimeter System

The RPS consist of two vehicles, a computer data van and a radar boom truck. The data van houses and transports a computer used for automatic control of the RDADS, data acquisition and storage, and computational support. Data van subsystems are all part of the main processor. These subsystems include the Central Processor Unit (CPU), program and data storage equipment, and the Operator's I/O console.

The radar truck is a flatbed truck with a tandem boom mounted on the bed. The boom is used to position the polarimeter radar equipment and antennas over a test site. Subsystems of the radar boom truck include the radar, the Antenna Positioning System (APS), and the Remote Data Acquisition and Distribution System (RDADS). The radar subsystem acquires polarized microwave backscatter measurements. Antenna orientation and position are controlled by the APS. The radar and APS are both controlled by the RDADS. Moreover, the RDADS acts as the smart interface between the RPS subsystems and the data van computer.

Because of the nature of the radar polarimeter and the measurements to be made with it, the radar antennas must be translated over an azimuthal arc during operation. This azimuthal scan is accomplished by turning the boom at its base. In addition, data must be acquired at different elevation angles in order to observe the effect of incidence angle on radar backscatter measurements. Figures 1 and 2 present drawings of the radar boom truck from two different views to illustrate the two angular freedoms of movement. Each of these angular position parameters (azimuth and elevation) is controlled by the RDADS through the APS interface hardware.

Microwave measurements are made by radar hardware mounted on the boom truck. The radar is composed of the RF, IF, and video electronics required to make the microwave backscatter measurements. The radar operates at frequencies in three different bands (X, L and C). This allows effects of frequency on target sites to be studied. Individual, dual-polarized transmit and receive antennas for each microwave band are part of the radar's RF equipment. These antennas allow measurements to made for four different combinations of transmit and receive polarization. IF and video electronics are used for range gating and integration of radar measurement signals. All radar circuitry is controlled and monitored by the RDADS.

Figure 3 presents a block diagram of the entire Radar Polarimeter System showing each of the subsystems mentioned above. This document describes the design and development of the RDADS. The next section covers the interface and operational requirements of the system. The design and development are then covered in two sections, one on

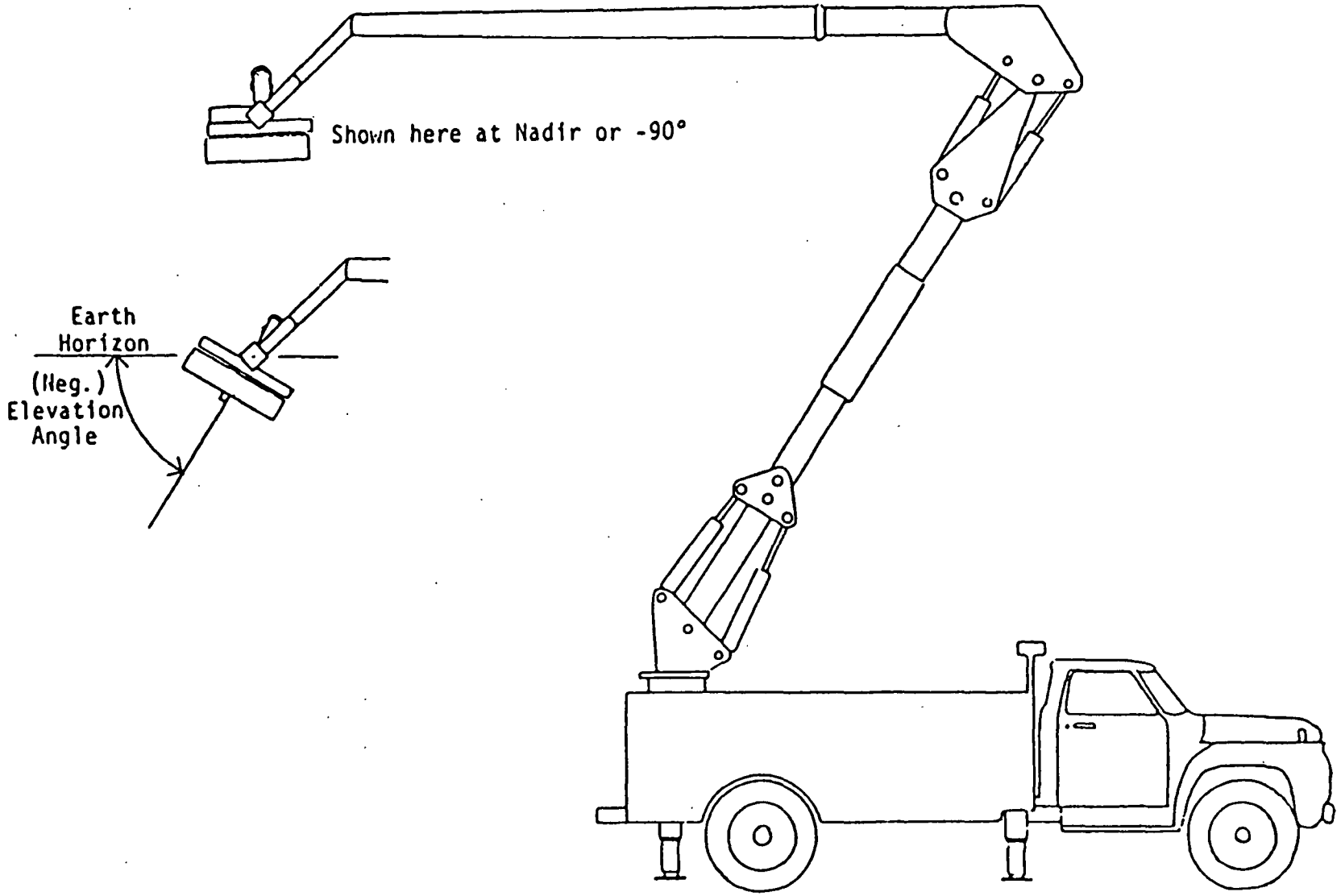


Figure 1. Elevation Angle of Radar Polarimeter Antennas.

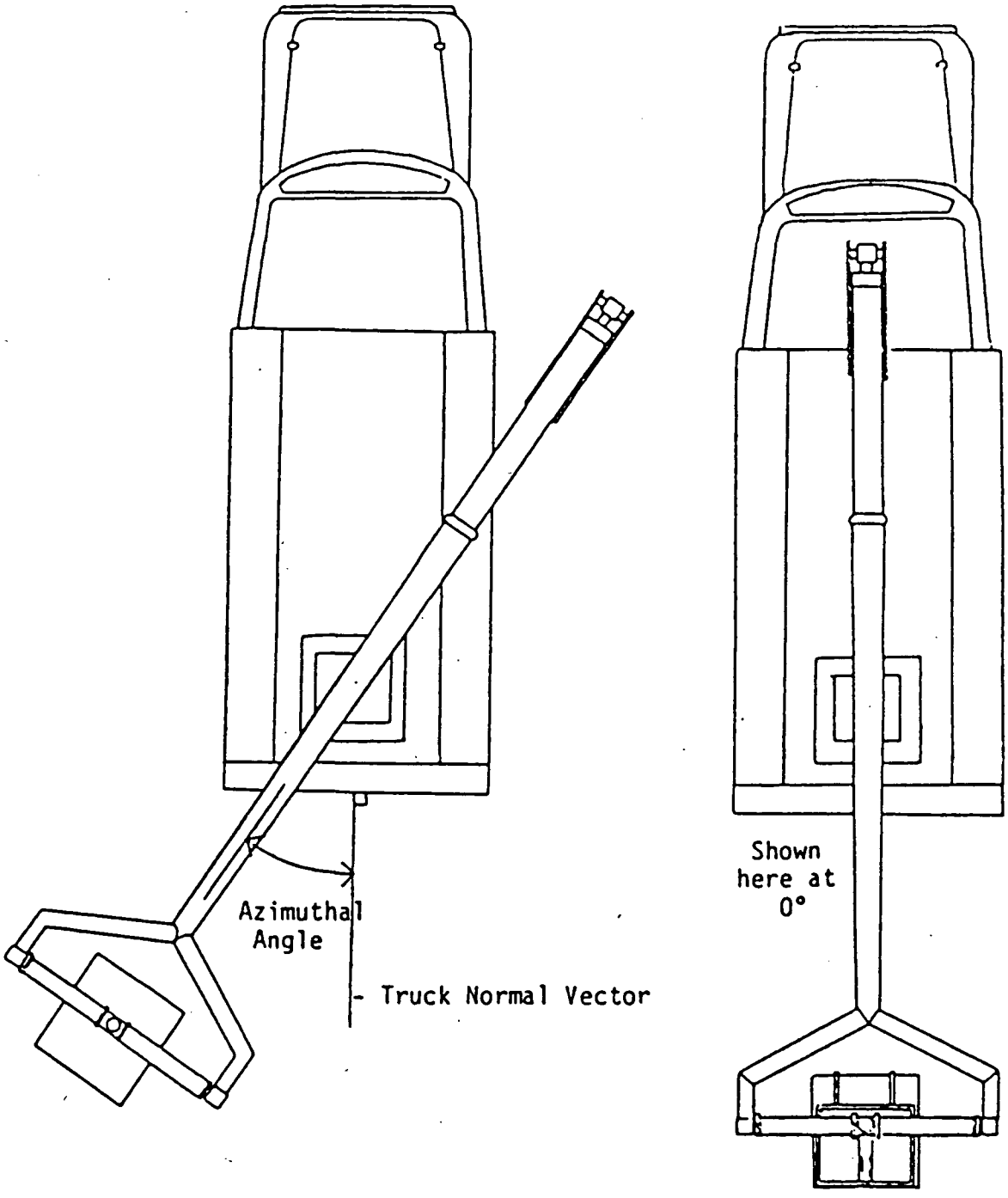


Figure 2. Azimuthal Angle of Radar Polarimeter Boom.

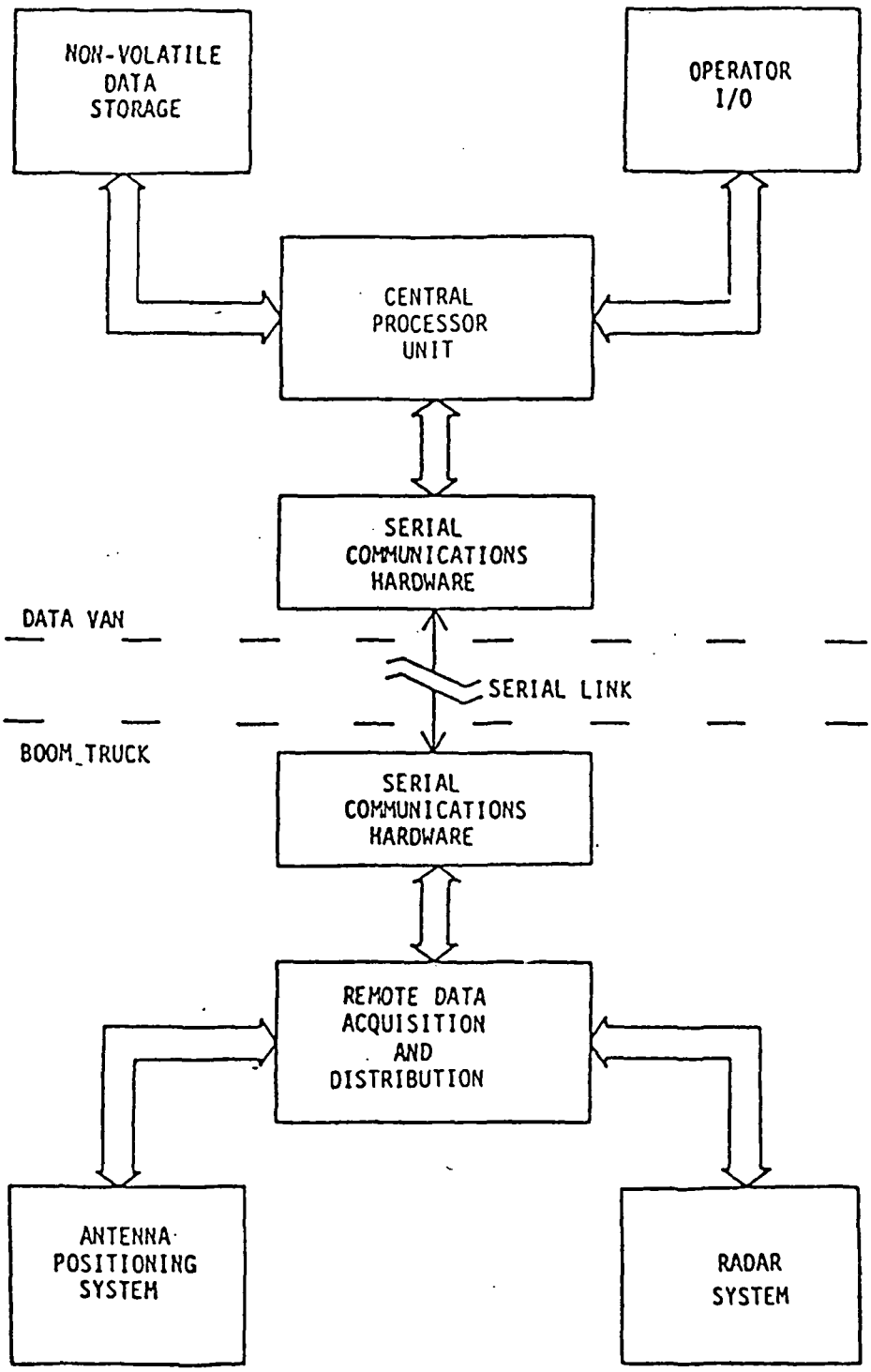


Figure 3. Block diagram of the Radar Polarimeter System.

hardware and one on software. A final section summarizes the design and recommends future work.

STATEMENT OF WORK

The objective of this project was to design, implement, and document the Remote Data Acquisition and Distribution System (RDADS). This effort included the hardware required to interface and control the radar equipment and the APS electronics. Hardware was also required for communicating with the main processor. The work also included all software required to operate the radar and to communicate with the main processor. Development of the APS hardware and the RDADS software used to control it were not a part of this work. The development of radar RF, IF, and video electronics were also not a part of this project.

The primary function of RDADS was to control and monitor the functions of the radar electronics. The system allows for automatic operation of the radar under the data van computer's direction. This section will review the radar electronics of the RPS to identify the interface and control requirements for the RDADS.

Radar Electronics

Radar Hardware mounted on the boom truck accomplishes the actual microwave measurements performed by the RPS. This hardware consists of three separate microwave transceivers multiplexed into a single Intermediate Frequency (IF) processing section. Each RF head operates at a specific frequency, namely 1.6 GHz for L-band, 4.75 GHz for C-band, and 10.0GHz for X-band. The radar system uses a pulse compression technique to achieve high range resolution. The IF section generates an

FM signal centered at 60 MHz. This phase coded expanded pulse is up converted to microwave frequencies in each radar head. The IF section also accepts the signal received by the RF head once it has been converted back to the 60 MHz IF range. Pulse compression, video amplification, range-gating, and integration are performed in sequence on the return signal by the IF receiver. The RDADS computer controls and monitors the operation of these radar components. Figure 4 presents a block diagram of the radar hardware.

Microwave Transceivers

The RF heads transmit electromagnetic microwave energy to the target through dual polarized antennas. Identical antennas are used to receive the microwave energy reflected by the target. For each of the microwave transceivers there is a pair of antennas, one transmit and one receive. Each antenna has two coaxial feed connections, one for vertical polarization and one for horizontal polarization.

A mechanical microwave switch is used to select which feed, and therefore which polarization, is used for the transmit and receive antenna. These switches, referred to as Polarization Transfer Switches (PTS), are controlled by the RDADS. They are controlled by a single TTL level signal such that a high level selects the vertical feed and a low level selects the horizontal feed.

A four-port microwave switch is used in each RF head to allow an internal calibration measurement to be made. This Calibration Transfer Switch (CTS) is used to bypass the head's antennas by feeding the transmit signal directly into the receiver circuit. This switch is

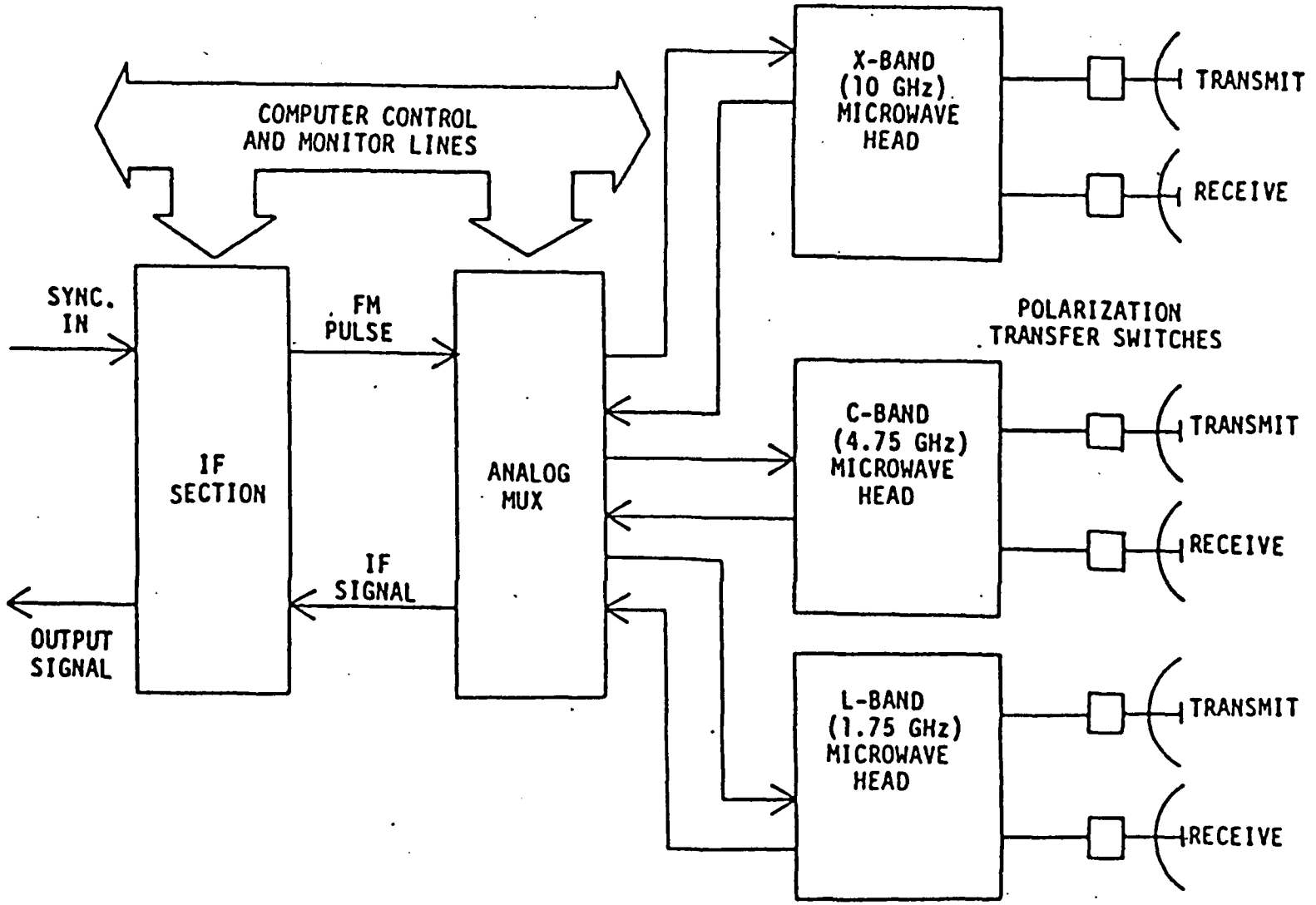


Figure 4. Block Diagram of Radar Hardware

controlled by the RDADS through two TTL signal lines. A pulse on one signal line causes the CTS to switch to the internal calibration loop. A pulse on the other line switches the CTS to allow transmission through the antennas. Each transfer switch has an internal single-pole, double-throw indicator switch that operates in conjunction with the microwave switch. These electrical indicator switches are used by the RDADS to monitor the state of each Polarization Transfer Switch and Calibration Transfer Switch.

Placement of the RF switches in the microwave circuitry is shown in Figure 5. This figure is a circuit diagram for a single RF head. The microwave source for each head determines its frequency of operation. The individual components in each head are either broadband devices or designed for operation in the frequency range of that head.

IF Section and Multiplexer

The IF section of the radar generates a swept frequency pulse that is directed to a selected RF head through the analog multiplexer. This FM pulse is then up converted by the selected head to the proper RF frequency and transmitted to the target. The reflected signal received by the RF head is down converted back to the IF range. The IF section will receive this down converted signal via the analog multiplexer and estimate the return energy. An analog signal whose voltage level is a function of return energy is output by the IF section.

Figure 6 shows a block diagram of the IF section of the radar. The transmitter portion generates the swept frequency pulse that is centered at 60 MHz. A square pulse of approximately 50 nS duration is

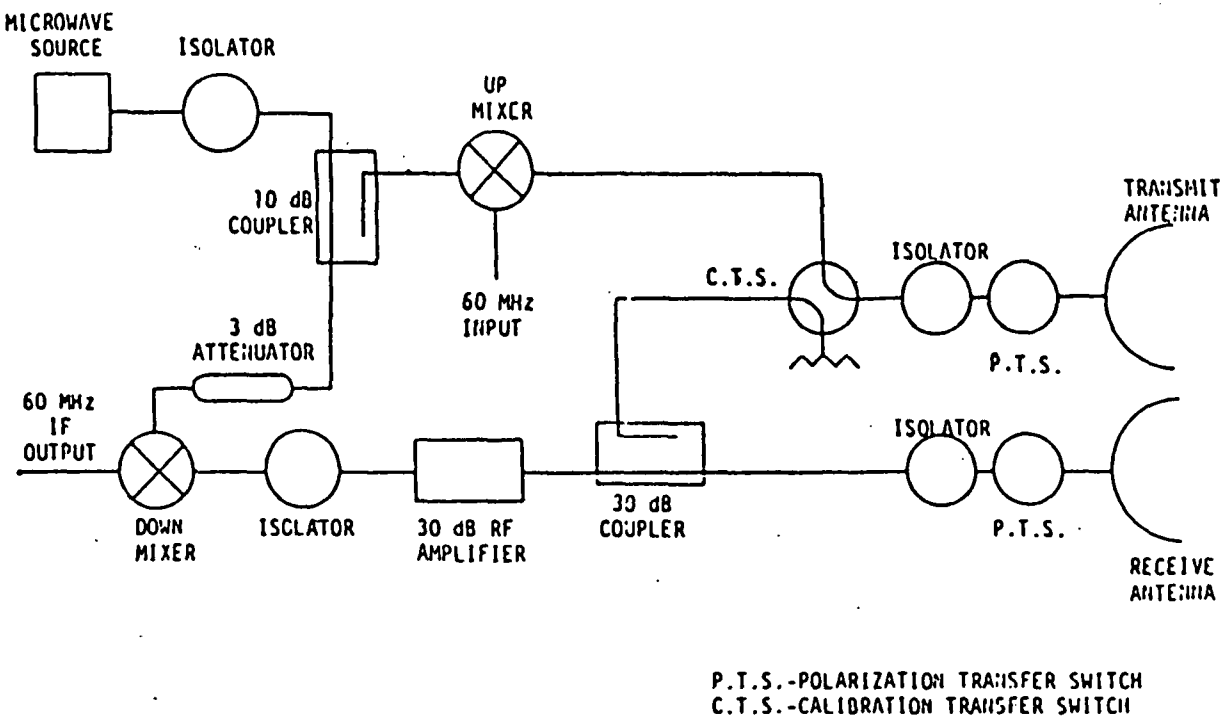


Figure 5. Microwave Transceiver Circuit Transfer Switches Controlled by RDADS.

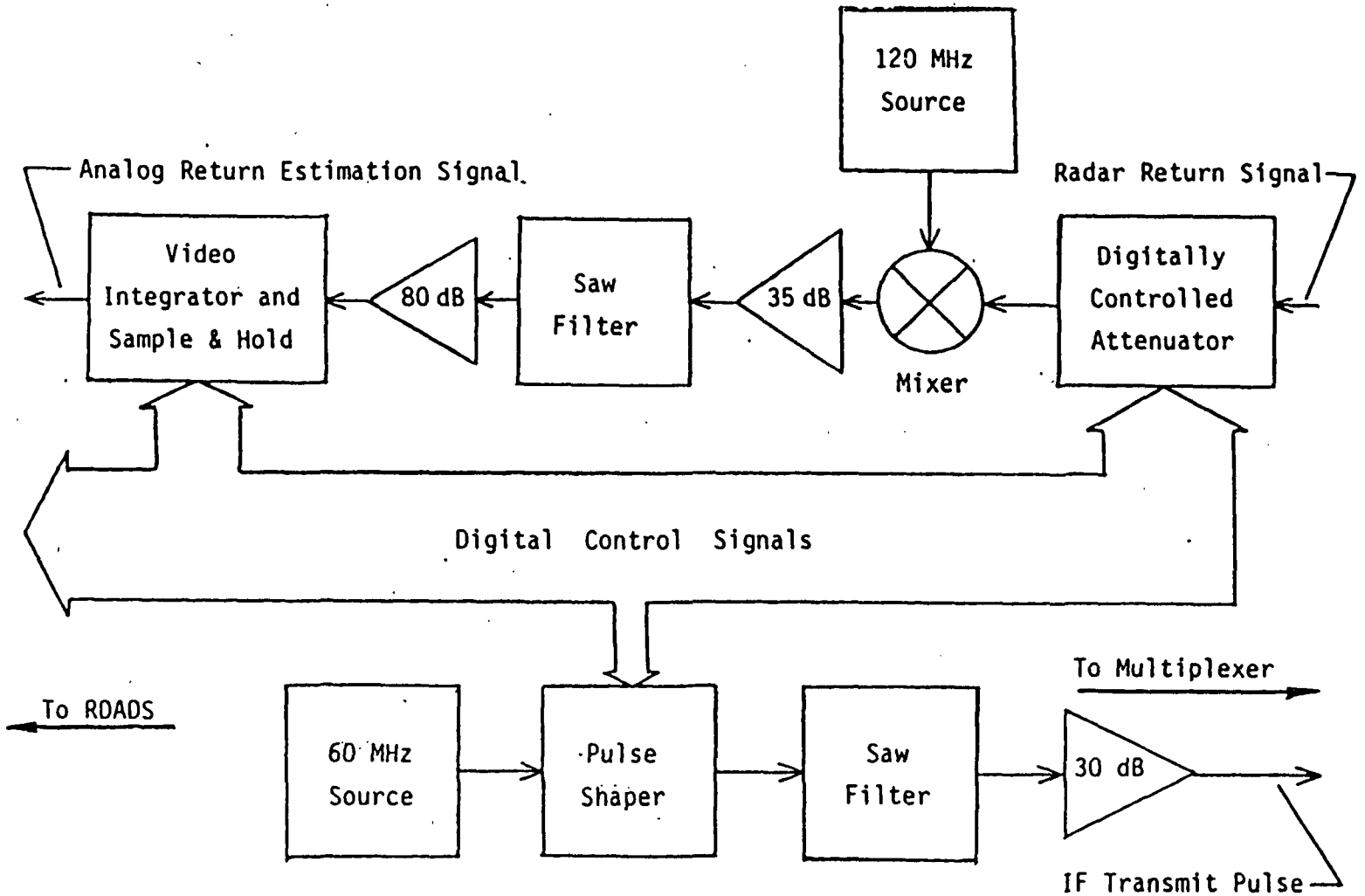


Figure 6. Block Diagram of IF Section of the Radar

required as an input to the IF transmitter. This pulse is used to generate a 60 MHz tone burst of the same duration. When the 60 MHz pulse is applied to the SAW filter device a Frequency Modulated (FM) pulse is produced. This swept frequency pulse is then upconverted by the RF transmitters.

The receiver portion of the IF section requires several input signals for proper operation. The 7-bit digital attenuator is used to control the gain of the IF receiver. This attenuator is controlled automatically by the RDADS. A range-gating pulse must be input to the IF receiver to control the gating of received signals into the Video Integrator. By providing the range-gating signal at the proper time and duration, the receiver can reject unwanted reflections and noise. The time of occurrence of the gate signal with respect to the occurrence of the IF transmit pulse will vary depending on the distance to the target and for internal calibration measurements. In addition, the duration of the gate signal varies with changes in the incidence angle of the electromagnetic energy on the target. The RDADS provides the range-gating signal at the proper time and of the proper duration, dependent on elevation angle and when the RF head is in calibrate mode.

The RDADS also supplies a "sample-and-hold" control signal and an "integrator reset" signal to the IF receiver. These signals are applied to allow integration of several received pulses and sampling of the resulting integrator output which is held as the IF section's analog output. The RDADS measures the analog output by converting it to a digital count.

The analog multiplexer used to select between radar heads uses mechanical switches identical to those used for polarization control. Two switches with the common port of one connected to one of the switched ports of the other are used to multiplex three lines to one. One switch pair is used for switching the IF transmitter's output to one of the three RF heads. The return signal of each head is connected to another switch pair to multiplex these into the IF receiver. Two TTL level signals, one for each switch, are required for each pair. These switches also have indicators that are used to monitor their state. The RDADS controls and monitors both multiplexer switch pairs.

Other RDADS Interfaces

The RDADS must also provide interface capability for use by the APS. Both analog input and output are required for the control and monitoring of antenna position. Analog outputs are used to signal D.C. motor control drive circuits to set the motor speed. Angular position information is input as an analog signal. Digital control lines are also used by the APS to control direction of movement and braking functions. The RDADS has provisions for these analog I/O and digital output signals.

The RDADS computer communicates with the data van computer through a serial RS232 data link. The RDADS has an asychronous communications channel used for this purpose. The main processor sends instructions for controlling the operations of the RPS to the RDADS through this link. The RDADS provides automatic functions that support and simplify the control operations. Data obtained by the RDADS are transmitted to the data van over the same serial data link. D.C. power for the boom

truck sub-systems is provided by the RDADS. This power is generated using D.C. power supplies that operate off of a 115 volts 60 Hz power source.

The interface and operational requirements of the RDADS have been identified. Of these, the most important is the radar interface since it is the means by which the polarimeter measurements are made. Provisions were identified for interfacing to the APS hardware developed external to this project. Operations controlled by the RDADS are done so under direction of the main processor in the data van. Instructions from the main processor and data returned by the RDADS are passed through a serial data link between the two systems. The next section will describe the hardware used to implement the RDADS.

HARDWARE

This section presents the design and implementation of the RDADS hardware. A short presentation of the design philosophy adopted for this project will be given. Following this, a complete description of the hardware installation will be given with details on interfaces to other RPS subsystems.

Design Approach

To obtain the best time wise execution of this project and still maintain high degrees of reliability and flexability, proven components and technology were utilized wherever possible. Industrial grade components were chosen from available "off-the-shelf" hardware to meet the RDADS operational requirements. These Original Equipment Manufacturer (OEM) designated products simplified the design by providing flexible, high level functions. Whenever a requirement could not be met with available industrial products, an original design was implemented.

A single board, microprocessor based, computer is the center of the system's hardware. This computer controls the operations of the RDADS under direction by the data van computer. The RDADS computer board also provides capability for interfacing directly to other RDADS, Radar, and APS hardware. Since the design uses a microprocessor, most of the system's operation was defined in software. This allowed a large degree of flexibility in the design. In addition, operations

that would be complex or difficult to implement in hardware were more easily accomplished in software. Other OEM products utilized in the design were analog I/O boards, a memory expansion board, a cardcage providing a bus backplane, and several D.C. power supplies. Collectively, these components were labeled the RDADS Computer.

The range gating function of the IF receiver presented a special design problem for the project. The required speed of operation for a controller could not be met using any available OEM products. An original design utilized Schottky TTL integrated circuit devices (ICs) to accomplish high speed control functions. The design was implementnd on a single printed circuit board labeled the IF Controller or IFC. This circuit also expanded the RDADS Computer's parallel I/O capability in the IF section.

RDADS Computer

The Remote Data Acquisition and Distribution System computer controls the operation of boom truck subsystems. This stand alone processor has the following duties:

- o Communicate with the data van CPU.
- o Control the IF and RF hardware.
- o Control the antenna elevation and boom scan.
- o Acquire radar data.

Interfaces to other RPS subsystems are an integral part of the RDADS hardware. These interfaces required the computer to have both digital and analog I/O capabilities. The hardware of the RDADS

computer can be broken down into the following components:

- o Central Processor with digital I/O capabilities.
- o Analog I/O Interface.
- o Random Access Memory.
- o Chassis and Power Supply.

Selections made for each of these components will be discussed in the following sections.

Central Processor

The central processor controls all activity of the RDADS. Intel Corporation's iSBC-80/24 Single Board Computer System was chosen for use as the central processor. The 80/24 is a single printed circuit board that uses an Intel 8085A-2 microprocessor. Since the 80/24 is designed for general applications, it is very versatile and can be configured in a number of different ways. The board has many jumper selectable component interconnects and set-up options. Several I.C. components on the board have software programmable functions. The 80/24 also accommodates connection directly to special configuration circuit boards.

The 80/24 is compatible with the IEEE-796 Bus Standard (known commonly as the Multibus). The board provides a system clock, parallel data communications lines, a serial communications channel, firmware (EPROM) sockets, programmable interval timers and event counters, and interrupt control circuitry. General specifications for the iSBC-80/24 are as follows:

- o Complete computer system on a single 6.75 by 12.0

inch printed circuit board.

- An Intel 8085A-2 microprocessor operating at 4.8 MHz.
- Sockets for 4K bytes (using 2708 or 2758), 8K bytes (using 2716), 16K bytes (using 2732), or 32K bytes (using 2764) of EPROM.
- 4K bytes of Random Access Memory on board.
- Multimaster-Multibus bus arbitration logic.
- Bus addressable to 64K.
- USART with RS232C line drivers and receivers for serial I/O up to 38,400 baud.
- Three programmable BCD or binary timers and event counters.
- 19.35, 9.68, 4.84, 2.15 and 1.075 MHz on board clock signals.
- Programmable control logic for up to 12 levels of vectored interrupts.
- Forty-eight programmable parallel I/O lines with sockets for I/O line drivers and terminators.
- Two iSBX bus Multimodule interface connectors.

For complete details on operation and use of the board see the "iSBC-80/24 Single Board Computer Hardware Reference Manual" [11].

The 8085 microprocessor controls all functions of the 80/24. The board was configured for installation of up to four 2732 EPROMs through jumper selections. Three 2732 EPROMs were used to store a control program for the 8085. This resident software directs all operations of

the RDADS. It utilizes the 4K on-board RAM of the 80/24 for variable storage. The 64K bus addressing capability of the 80/24 was used to install an expansion RAM board. With this set-up, the processor has up to 16K bytes of constant program memory and 48K bytes of dynamic read/ write storage.

Serial I/O, Timers, and Interrupts

The serial channel of the 80/24 was used for communicating with the Data Van CPU. This channel was configured for a standard RS232 set up so that no special interfaces were required for connection of the serial data link. On-board line drivers and receivers convert 0 to +5 (TTL) voltage levels to and from the standard RS232 +12 to -12 voltage levels. The serial interface was set up for loop-back of all RS232 hand-shaking signals, thus it required only three connections. These were Transmit Data (TxD), Receive Data (RxD) and a signal ground. Serial communication is controlled by an 8251A Programmable Communications Interface (PCI) I.C. The 8085 sends and receives data through the 8251 in a parallel format. Conversion of the parallel data to and from serial data formats is handled by the 8251 PCI.

An 8253 Programmable Interval Timer (PIT) chip on the 8024 provides three independent programmable timers. Timer 2 of this device is used as a baud rate generator for the serial communications channel. The input to Timer 2 is the on-board 1.075 MHz clock signal. The timer is programmed for square wave, divide by 7, operation so that it outputs a 153.6 KHz clock signal. This signal is applied to both the receive and transmit clock inputs of the 8251 PCI. The 8251 was

programmed for an asynchronous communications rate multiplier of 16 so that the baud rate of the serial interface is 9.6 KHz or 9600 Baud. Since both the 8251 and the 8253 are software programmable the baud rate for the system can easily be adjusted.

The 1.075 MHz clock signal was also input to Timer 0 of the 8253 PIT. This timer was programmed to divide the input by 53,750 in order to produce a pulse output rate of 20 Hz. The output of Timer 0 was tied to the IR1 pin of the 8259A Programmable Interrupt Controller (PIC). This provided a Level 1 Hardware Interrupt every 50 mS for timing various system functions. Again, since the PIT is software programmable, this real time clock rate can be modified.

An output from the IF section is used to syncronize the RDADS processor's data acquisition with the analog output from the radar. This output was the IF Service Request (IFSR) signal. IFSR tells the processor when the analog output of the receiver is ready for conversion. To reduce the radar data rate, IFSR is input to Timer 1 of the 8253. Timer 1's output was then applied to IR2 of the 8259A PIC. The timer was programmed for divide by 21 operation. IFSR cycled at a fixed rate of 2.5 KHz so that a Level 2 interrupt occurred every 8.4 mS when enabled. During radar data acquisition the Level 2 interrupt triggers the analog data conversion done by the processor. The analog data acquisition rate is approximately 119 samples per second. The software that programs the operation of Timer 1 of the 8253 can easily be changed to obtain a different data acquisition rate.

Two other interrupt levels are used in the RDADS processor. These are utilized for interrupt driven serial I/O: Level 6 interrupt for

serial input and Level 7 for serial output. The Receiver Ready (RxR) signal output by the 8251A PCI was tied to IR6 of the PIC. Transmitter Ready (TxR) from the PCI was connected to IR7 of the PIC. With these two interrupts, the processor did not have to poll the PCI during I/O operations. Instead the processor could send data to the 8251A and then do other work while the 8251A output the data. The processor could also be interrupted during non-critical operations when data was received by the 8251A.

Parallel I/O

Parallel I/O lines of the 80/24 were used for hardware control and monitoring. This parallel communications is accomplished on the 80/24 by two 8255A Programmable Peripheral Interface I.C.s. Each 8255A PPI has three 8-bit programmable I/O ports. Ports A and B can be programmed independently as either input or output. Port C is programmed independently as either input, output, half input and half output, or as handshaking signals for I/O operations of Ports A and B. Also, when programmed for output, signals from Port C can be controlled individually through bit set/reset commands issued to the 8255A's control port. The two 8255A's on board the 80/24 provide a total of 48 parallel I/O lines divided into six ports. Port addresses for all six ports are summarized in Table 1.

Ports E8, E9, and EA were used for communicating with the RF hardware. Port E8 was programmed as output and used to control the calibrate/operate (Cal/Op) transfer switches in the RF heads. A Cal/Op switch has two positions and requires two signals to control it. A

TABLE 1

Port Addresses for the 80/24's Parallel I/O (Hex)

8255A Port	8255A No. 1	8255A No. 2
A	E4	E8
B	E5	E9
C	E6	EA

pulse on the switch Control 0 signal would set the switch to the calibrate position. Likewise, a pulse on Control 1 would place the switch in the operate position. Each had to be controlled independently so that only one selected head would be in operation at any given time. Designating Port E8's eight bits for Cal/Op control allowed for connection to the three existing radar heads and one additional backup or expansion connection. The assignment of the output signals for Port E8 are given in Table 2.

The two control signals for a Cal/Op switch energize two respective coils internal to the switch. The switch assembly has an internal electronic circuit that automatically turns either coil off after it accomplishes its internal mechanical operation. This means that the controlling signal can be a level rather than a pulse which simplifies the algorithm used to drive the port. The only requirement is that the two signals applied to a Cal/Op switch must never both be high at the same time. This approach also supports the use of transfer switches that have a single level sensitive control input.

The radar head numbers specified in Table 2 were used during

TABLE 2

Cal/Op Control Port Signal Assignments

Port E8 Bit	Assignment
0	Head 1 Cal/Op Control 0
1	Head 1 Cal/Op Control 1
2	Head 2 Cal/Op Control 0
3	Head 2 Cal/Op Control 1
4	Head 3 Cal/Op Control 0
5	Head 3 Cal/Op Control 1
6	Head 4 Cal/Op Control 0
7	Head 4 Cal/Op Control 1

development when the actual microwave band assignment was of no concern. These head numbers were assigned to particular bands in the final system as follows:

Head 1 - X Band (10.0 GHz).

Head 2 - L Band (1.75 GHz).

Head 3 - C Band (4.75 GHz).

Head 4 - Reserved for future use.

This assignment was purely a matter of choice and had no impact on the design. It is presented here for clarity.

I/O Port E9 on the 80/24 was used in monitoring the status of all microwave switches in the radar heads. This port was programmed for output operation and used to control the multiplexing of switch state indicator circuits. Each switch assembly has an electrical single-

pole, double-throw indicator switch that is set according to the current state of the microwave switch. Output signals from Port E9 drive the common pole of individual indicator switches or pairs of switches. Within an RF head the common poles of both the Cal/Op and the Transmit Polarization (Tx. Pol.) switches are connected to the same output bit of Port E9. Each pair was connected to a different bit, one bit for each head. The common pole for each Receive Polarization (Rx. Pol.) switch is connected to other bits of port E9, again one for each head. These signals out of Port E9 are called the Indicator Circuit Enables and the specific assignments are given in Table 3.

TABLE 3

Indicator Enable Port Signal Assignments

Port E9 Bit	Indicator Circuit Assignment
0	Head 1 Tx. Pol. and Cal/Op
1	Head 1 Rx. Pol.
2	Head 2 Tx. Pol. and Cal/Op
3	Head 2 Rx. Pol.
4	Head 3 Tx. Pol. and Cal/Op
5	Head 3 Rx. Pol.
6	Head 4 Tx. Pol. and Cal/Op
7	Head 4 Rx. Pol.

The upper half (Bits 4 through 7) of Port EA was programmed for input and used to read the status of the microwave switches. Assignments for these input lines are given in Table 4. Two are used for polarization switch indicator monitoring and two for Cal/Op switch indicator monitoring. Each pair consist of one line labeled Indicator 0 and one line labeled Indicator 1. The polarization indicator pair connects to both the Tx. Pol. and Rx. Pol. indicator switches in every radar head. Likewise, the Cal/Op indicator pair connects to the Cal/Op indicator switch in each head. All connections to indicator switch terminals are made through a signal diode, one diode for each terminal, with the anode end tied to the terminal. The cathode end of the diode ties to either Indicator 0 or 1 dependent on how the switch's RF ports are connected. The upper four bits of Port EA are pulled to a high level through 1K ohm resistors on-board the 80/24. Indicator circuits for a single radar head are illustrated in Figure 7.

TABLE 4

Indicator Circuit Read Port Assignments

Port EA Bit	Indicator Read Function
4	Polarization Indicator 0
5	Polarization Indicator 1
6	Cal/Op Indicator 0
7	Cal/Op Indicator 1

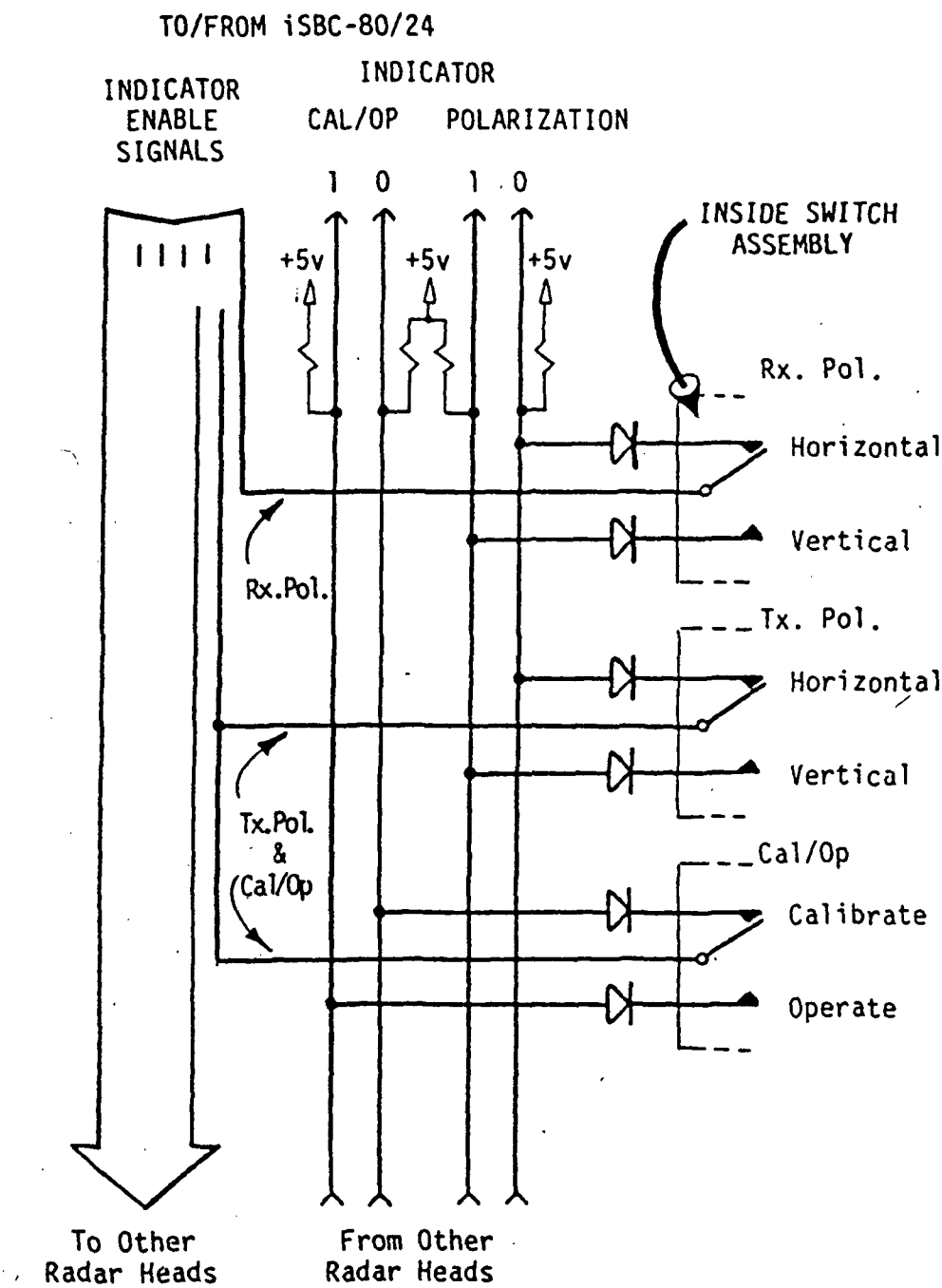


Figure 7. Indicator Circuits for a Single Radar Head

Normally, the Indicator Circuit Enable outputs are at a high level. An indicator circuit can be read by the processor to determine the status of any switch in the radar heads as follows:

- o The Indicator Circuit Enable output for the desired head and desired switch (or switch pair) is driven low. Only one output of Port E9 should be low at any time or results of the read are unspecified.
- o The Indicator Read Port (upper half of Port EA) is input to the processor. The status of the selected switch is then determined from the data read.

The Indicator Read data can be divided into two separate 2 bit binary values, one for polarization and one for Cal/Op. Indicator 0 would be the least significant bit of the 2 bit value and the value would range from 0 to 3. Table 5 defines the meaning of data returned by an indicator circuit read. This table shows an open circuit failure as Failure I and a short circuit failure as Failure II.

The lower half of Port EA was programmed for output operation. Bits 2 and 3 were used to control the transmit and receive polarizations. Polarization switches required a single TTL level control signal. The microwave antennas were connected so that a low level signal corresponded to horizontal polarization and a high level signal corresponded to vertical. Since only one RF head can be operating (transmitting) at a time, the polarization transfer switches in each head were controlled simultaneously. This was accomplished using the two signals (bits 2 and 3) of Port EA. Bit 2 controls all transmit

TABLE 5

Interpretation of Indicator Circuit Data

INDICATOR LINE		INDICATOR PAIR	
1	0	Cal/Op	Polarization
L	L	Failure II	Failure II
L	H	Calibrate	Horizontal
H	L	Operate	Vertical
H	H	Failure I	Failure I

polarization switches and Bit 3 controls all receive polarization switches. The two remaining output bits of Port EA were also routed to every radar head. These bits provide spare common control signals to the RF heads reserved for future expansion or modification. The bit assignments of the lower half of Port EA are summarized in Table 6.

TABLE 6

RF Head Common Control Port Signal Assignments

Port EA Bit	Control Function
0	Reserved RF Control 0
1	Reserved RF Control 1
2	Transmit Polarization
3	Receive Polarization

Parallel Ports E4 and E6 of the 80/24 were used for monitoring and control of the IF section. Port E4 was utilized as a bidirectional data port for I/O operations to the IF Controller board. Port E6 was programmed for output and used primarily to control the I/O operations of Port E4. Both of these ports and their assignments will be discussed in more detail in the second part of this section that covers the IFC.

Port E5 was reserved for use by the Antenna Positioning System. Interfaces to the APS were developed external to this project. For information on the use and assignment of signals of Port E5, documentation on the APS should be consulted. The use of all parallel I/O ports on the 80/24 is summarized in Table 7.

Analog I/O Interface

A unique feature of the 80/24 is that it is Multimodule compatible. The two iSBX bus interface connectors allow two iSBX Multimodule boards to be mounted directly on the 80/24 board. The 80/24 can be specially configured for different applications through the use of available Multimodule boards. Intel currently supplies several Multimodule boards for a variety of applications. Two such boards where obtained for analog I/O functions.

This part of the RDADS processor actually consists of two components: an analog input board and an analog output board. Each of these boards are Intel Multimodule boards that mount directly on the iSBC-80/24 board. This requires the use of both of the 80/24's Multimodule connectors.

TABLE 7

iSBC-80/24 Parallel I/O Summary

Port Address	Assigned Function
E4	IFC Data I/O
E5	Reserved (APS)
E6	IFC I/O Control
E8	Cal/Op Control
E9	Indicator Control
EA Lower	RF Head Control
EA Upper	Indicator Read

The analog input board selected for use in the RDADS was the iSBX-311 Analog Input Multimodule board. The specifications for this board are as follows:

- o Eight differential or sixteen single-ended analog inputs.
- o Resistor selective gain (1X-250X).
- o -5 to +5 volt bipolar or 0 to +5 volt unipolar input range.
- o 12 bits Resolution (11 bits plus sign for +/- volts).
- o Accuracy of +/-0.035% of full scale range +/- 1/2 LSB.
- o 20 megohms input impedence with input protection to 30 volts.

- o Typical conversion time of 50 microseconds.
- o 18K conversions per second sample rate.

The iSBX-311 was used in its factory configuration of single ended, bipolar operation with a 1X gain setting. This set-up provided 16 analog inputs with an input range of -5 to +5 volts and input resolution of 2.44 millivolts. Complete details on the iSBX-311 can be found in the board's hardware reference manual [12].

The analog inputs provided by the 311 are used to measure the radar output signal, antenna elevation angle, and supply voltage levels. Some of the inputs to this board were left unused and are reserved for future expansion or modifications. The signal assignments for the 311 are itemized in Table 8. All D.C. supply voltage levels were divided down using trimming potentiometers (trimpots) to produce the monitor levels input to the analog converter. The 117 volt A.C. level was conditioned using a low voltage transformer, single wave rectifier, and R-C filter. This produced a D.C. voltage level corresponding to the average peak magnitude of the A.C. line voltage. The peak level signal was also divided down using a trimpot before input to the iSBX-311. The trimpots are adjusted to calibrate supply voltage monitoring.

The iSBX-328 Analog Output Multimodule board was chosen for use as the analog output interface. The 328's specifications are listed below:

- o Eight independent analog voltage level or current loop (individually selectable) outputs.
- o Output ranges of -5 to +5 volts bipolar level, 0 to

TABLE 8

iSBX-311 Analog Input Signal Assignments

Channel	Input Signal
0	IF Analog Data Signal
1	APS Position Signal 1
2	APS Position Signal 2
3-6	Reserved (Not Connected)
7	117 Volt A.C. Monitor
8	Reserved IF Analog Signal
9	+5 Volt Supply Monitor
10	-5 Volt Supply Monitor
11	+15 Volt Supply Monitor
12	-15 Volt Supply Monitor
13	+12 Volt Supply Monitor
14	-12 Volt Supply Monitor
15	+28 Volt Supply Monitor

+5 volts unipolar level, and 4 to 20 milliampere current loop.

- 12 bit resolution.
- Accuracy of at least 0.17% of full scale voltage range.
- Output slew rate of 0.1 volt per microsecond minimum.
- 5 KHz single channel and 800 Hz eight channel

through put rates.

- o Voltage mode current output +/- 5 milliamperes maximum.

This board provides analog output for APS motor control functions. All outputs are reserved for use by the APS and future modification or expansion. For details on the operation and use of the iSBX-328 consult the hardware reference manual [13]. This project did not use any outputs from the 328.

Random Access Memory

Chrislin Industries' CI-8080 64K Dynamic Random Access Memory (RAM) board is used to provide the RDADS processor with additional program and data storage space. This board contains 64K bytes of RAM and is Multibus compatible. Due to on-board memory of the iSBC-80/24 the processor can address only 44K bytes of the CI board. The remaining 20K bytes can be used by other boards (added in the future) that are connected to the Multibus. Table 9 gives a breakdown of the 80/24's address space in terms of on-board memory and memory on the CI-8080.

TABLE 9

Memory Map for the RDADS iSBC-80/24 Processor

Address Range	Location	Storage Function
0 - 3FFF	80/24	Constant Program
4000 - EFFF	CI-8080	Dynamic Program & Data
F000 - FFFF	80/24	Program Variables

RDADS Chassis and Power Supply

The Multibus boards discussed above where installed in a card cage chassis. Electronic Solution's ESBC-604G was used for this purpose. The 604 is a four slot card cage with a Multibus backplane. It allows connection of a power supply directly to the bus backplane for power distribution to all cards installed in the chassis. Use of the four slots in the ESBC-604G was as follows:

Slot 1 - iSBC-80/24 Computer with two iSBX Multimodule boards mounted on it.

Slot 2 - Not available due to iSBX Multimodules on 80/24.

Slot 3 - CI-8080 64K RAM board.

Slot 4 - Available for future use.

Note that there is an extra slot available for future needs. The 604 also has an expansion connector on the bus backplane that allows an expansion card cage to be mounted directly to the 604.

Power is supplied to the card cage by Power One's CP-291A computer grade D.C. power supply. The 291 supplies all power needed by the 80/24, the Multimodule boards, and the CI-8080. Power from the 291 is also used by other subsystems of the RDADS. This supply outputs +/-5 volts and +/-12 volts. Three additional power supplies are used to provide +/-15 volts and +28 volts to the RDADS. Lambda's LUS-10-15 switching power supply is used for a +15 volt power source. Power One's HA15-0.5 D.C. power supply provides -15 volts. And Electro-static's Model 30-28 supplies +28 volts to the RDADS.

Figure 8 presents the component breakdown of the RDADS Computer. The figure shows how the components interconnect to each other and to other parts of the RDADS and RPS subsystems. The sections above have described the components of the RDADS computer. The next part of this section will present the IF Controller design.

IF Controller

A special function board was designed and developed by this project for controlling the radar range gating in the IF section of the RPS. The IF Controller (IFC) is a high speed (60 MHz) digital circuit with many programmable features. The IFC inherently controls the radar transmission and receiver data acquisition in the IF section. In addition the board provides the RDADS processor with expanded I/O capability in the IF section.

The IFC circuit board can be broken down into the following major subdivisions:

- o Processor I/O Interface
- o Timing Network
- o Cycle Control
- o Programmable Transmit Pulse Generator
- o Fixed Delay Generator
- o Programmable Range Delay Generator
- o Programmable Gate Pulse Generator
- o Digital Attenuator I/O Interface
- o Band Multiplexer Control Interface

Each of these IFC sub-circuits will be discussed in the next sections.

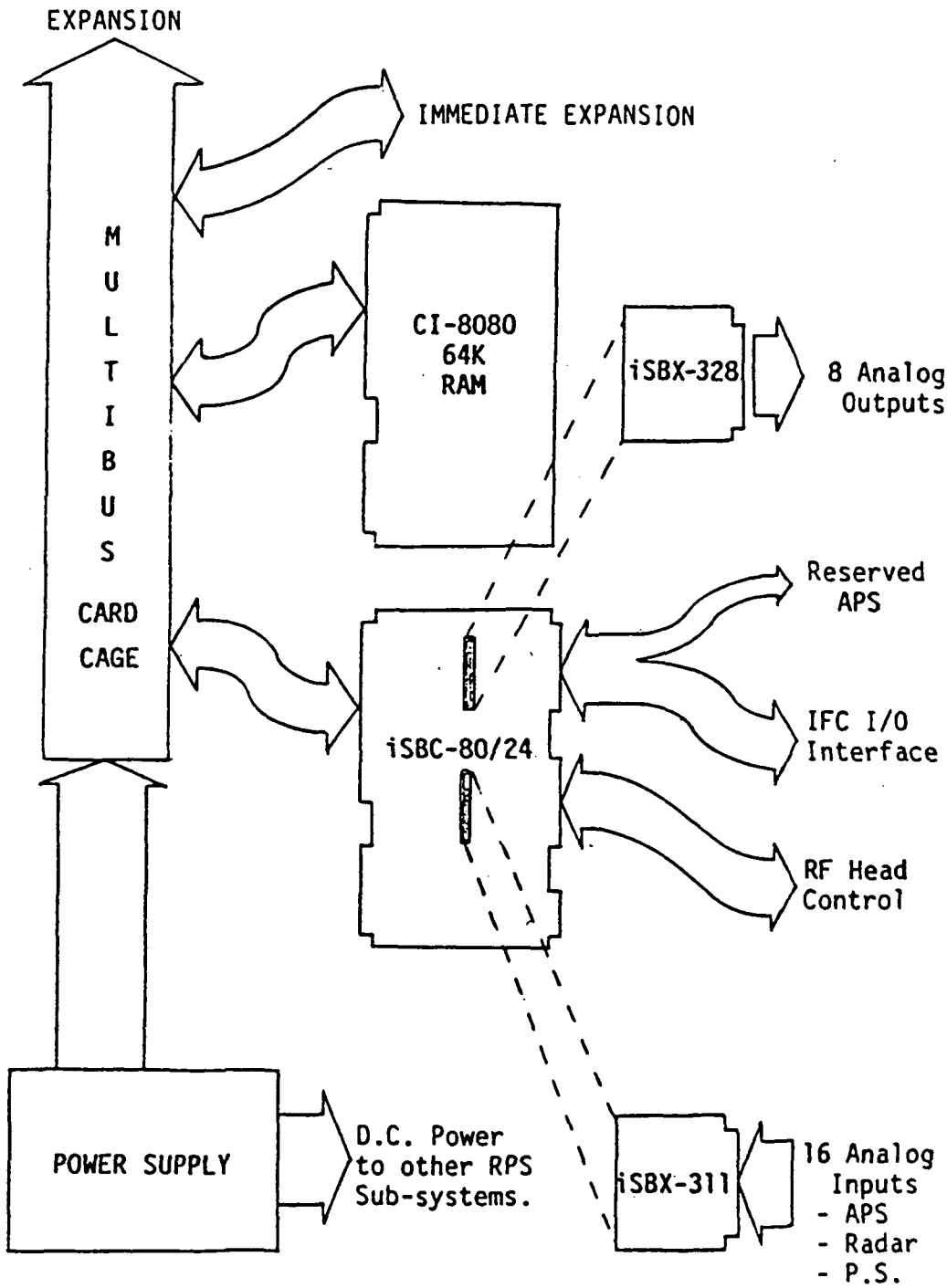


Figure 8. RDADS Computer Block Diagram.

Processor I/O Interface

As noted in the section on the RDADS processor's parallel I/O, Ports E4 and E6 are used for communicating with the IFC. Port E4 is used as a bidirectional bus port for data I/O to and from the IFC. Before the processor can perform an I/O operation through Port E4, it must first program the port for either input or output operation, whichever is appropriate. Port E6 is used to control the I/O operations to the IFC. Three bits of port E6 are used to select between different register pairs on the IFC board. Another bit is used to tell the IFC whether the processor is doing input or output. Two other bits are used to enable IFC output or strobe IFC data latching. Of the remaining two bits, one is used as a master reset output by the processor to the IFC and the other is reserved for future needs. Table 10 presents the IFC/Processor Interface parallel I/O assignments.

The names listed under "Signal" in Table 10 are the formal signal names used in the circuit diagrams. Processor communication to the IFC is either to a data latch or from a tri-state buffer. Latches and buffers are arranged as register pairs on the IFC and each pair is referred to as a channel. There is a total of 6 channels used in the circuit. The three select lines are used to choose one of the 6 channels. The IOC signal is used to select either the latch or buffer of the selected channel. These four signals combined are input into a 74154 4-to-16 Decoder I.C. on the IFC board. Six of the outputs of the 74154 are used as strobe signals to the six latches. Six others are used as output enable (OE) signals to the six tristate buffers. The remaining four outputs are not used. The PS/E and SS/E signals are

TABLE 10

Processor Port Assignments for IFC Interface

Port	Bit	IFC Function	Signal
E4	0	Data I/O - Bit 0	D0
E4	1	Data I/O - Bit 1	D1
E4	2	Data I/O - Bit 2	D2
E4	3	Data I/O - Bit 3	D3
E4	4	Data I/O - Bit 4	D4
E4	5	Data I/O - Bit 5	D5
E4	6	Data I/O - Bit 6	D6
E4	7	Data I/O - Bit 7	D7
E6	0	Data I/O Control	IOC
E6	1	Channel Select - Bit 1	SEL1
E6	2	Channel Select - Bit 2	SEL2
E6	3	Channel Select - Bit 3	SEL3
E6	4	Reserved Control Bit	n.a.
E6	5	Reset Signal	RST
E6	6	Secondary Strobe/Enable	SS/E
E6	7	Primary Strobe/Enable	PS/E

applied to the G1 and G2 enable inputs of the 74154. The outputs of the 74154 are all high until the device is enabled by the PS/E and SS/E signals. When enabled the selected output goes low to strobe or enable the proper device. With this configuration the IFC registers all share a common data bus connection to the processor through Port E4. Figure 9 details the I/O interface for the IFC. The assigned select addresses for each IFC channel are given in Table 11.

Channel 2 was used for general control in the IFC. Control information written to Latch 2 is used by various other parts of the IFC circuit. The outputs of this latch are also tied to the inputs of the tristate buffer it is paired with. The processor can read the contents of the latch through the buffer. Output bits from the latch are listed in Table 12 along with their assigned signal name. These signals are explained in the discussion on the sub-circuit identified in this table.

Timing Network

To provide synchronous operation with the other parts of the IF section, the IFC's clock signal is derived from the 60 MHz IF oscillator. The oscillator's output is brought on-board the IFC through an SMA connector. This signal is applied to a Schmitt Trigger Inverter to provide a clock signal of sufficient amplitude to the rest of the board. This signal is buffered once again to provide a clock signal of opposite phase. For lower frequency timing functions on the IFC, the buffered 60 MHz clock is sent through a set of cascaded 74S196 Divide-by-10 Counters. Three 74S196's are used to give a total clock

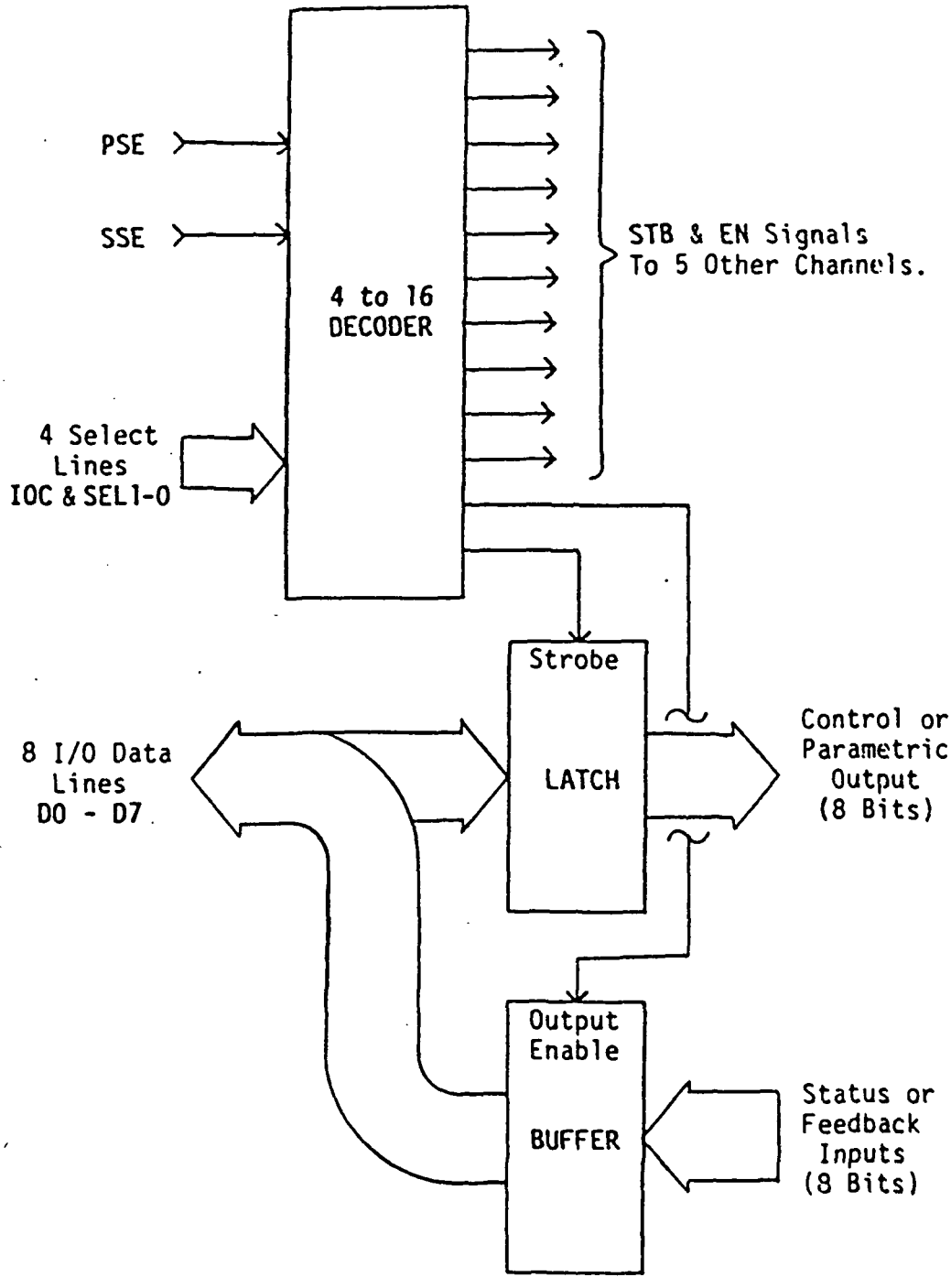


Figure 9. Processor I/O Interface on the IFC.

TABLE 11

IFC Channel Assignments

Channel	Register Selected
0-1	Not used.
2	General Control
3	Transmit Pulse
4	Range Delay
5	Gate Width
6	Digital Attenuator
7	Multiplexer

TABLE 12

IFC Control Channel (2) Signals

Bit	Signal	Sub-circuit
0	TNE	Timing Network
1	TxLL	Cycle Control
2	SRC	Cycle Control
3	FDA	Fixed Delay
4	RxLL	Fixed Delay
5-7	----	Not Used

frequency division of 1000. Each counter in the timing network has four outputs, all at a different frequency.

The 60 MHz buffered clock was labeled CK0 and the twelve outputs from the Divide-by-1000 counter were labeled CK1 through CK12 (CK1 = 30 MHz and CK12 = 60 KHz). A two phase 30 KHz signal (CK13) was produced by clocking a toggle Flip/Flop (F/F) with the 60 KHz CK12 signal. Figure 10 shows the timing network of the IFC. Note that CK1, CK9, CK10, and CK12 are the only clock signals output by the Divide-by-1000 counters that are used. Figure 10 also shows the introduction of the Reset (RST) signal into the IFC. This signal is buffered to provide reset signals to the timing network and other parts of the circuit. The Timing Network Enable (TNE) from the control latch, must be high for the circuit to operate.

Cycle Control

To control the cycling operations performed by other parts of the IFC, a Cycle Control circuit was designed. This circuit is presented in Figure 11. This circuit performs the following functions periodically:

- Tell the Transmit Pulse Generator when to load and when to run (TxSLC).
- Provide a service request interrupt signal to the processor (EOC).
- Tell the Analog Sample and Hold device when to sample and when to hold (S/H).
- Tell the Analog Integrator when to reset and when to integrate (R/I).

Note: (100K) ≙ 100KHz

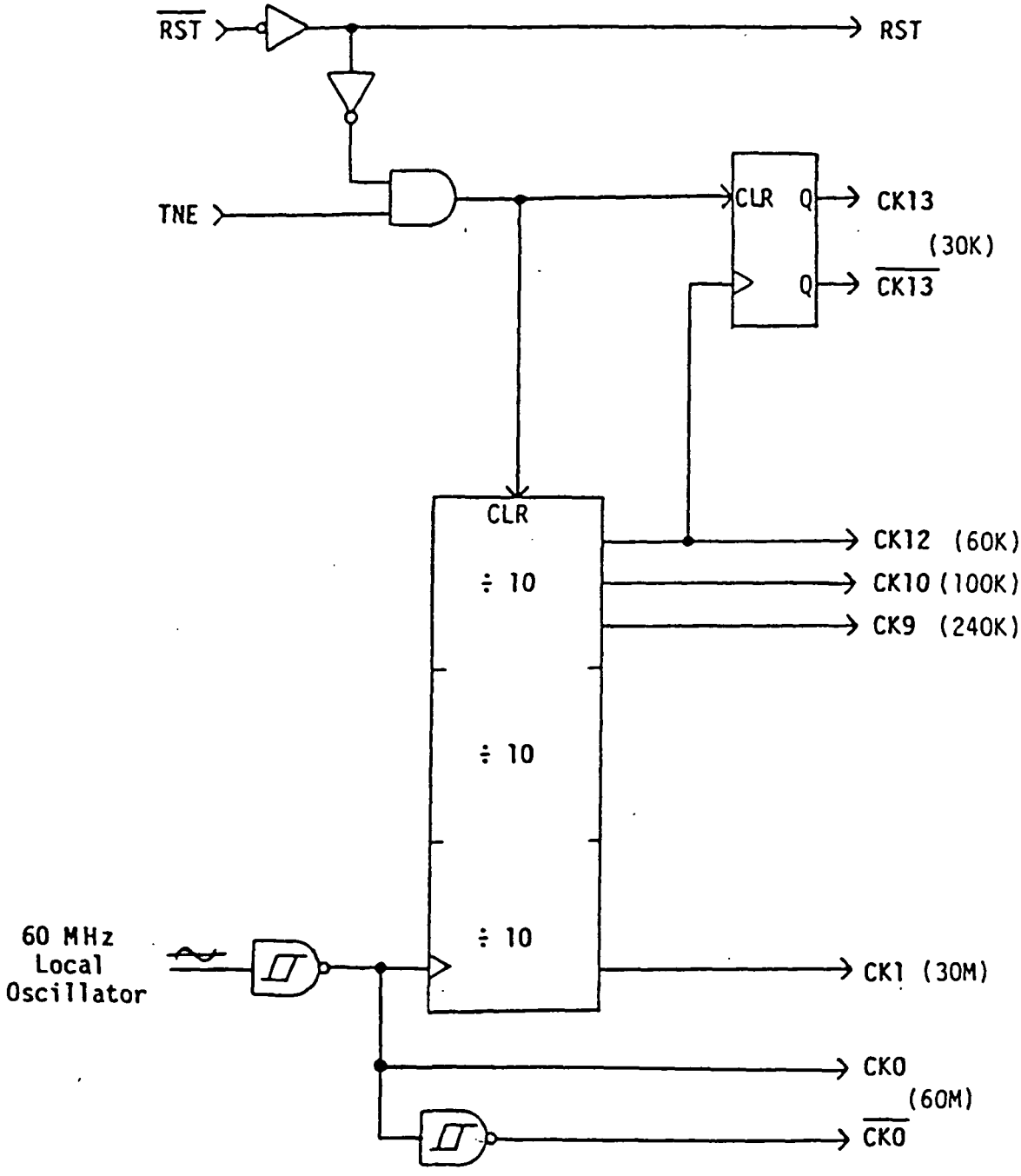
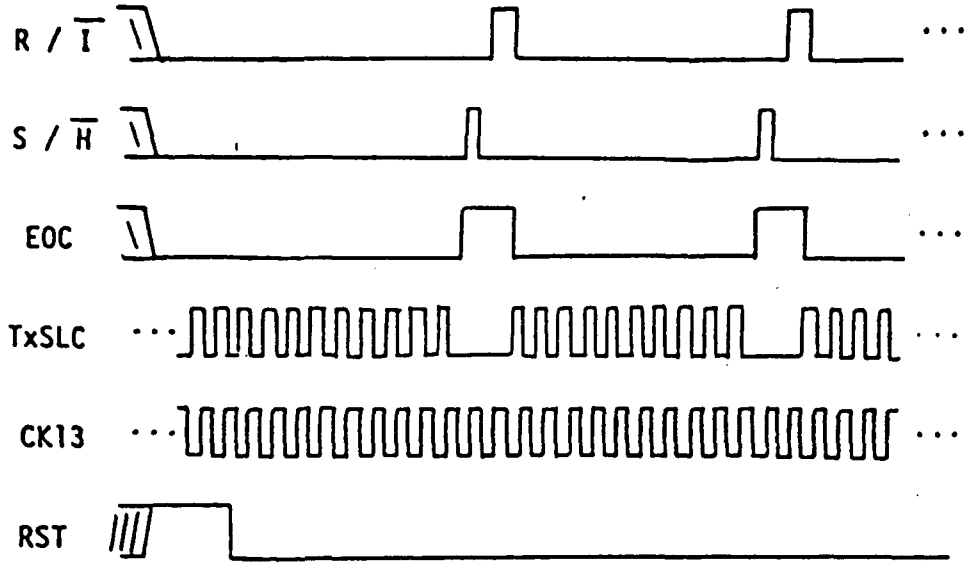


Figure 10. IFC Timing Network



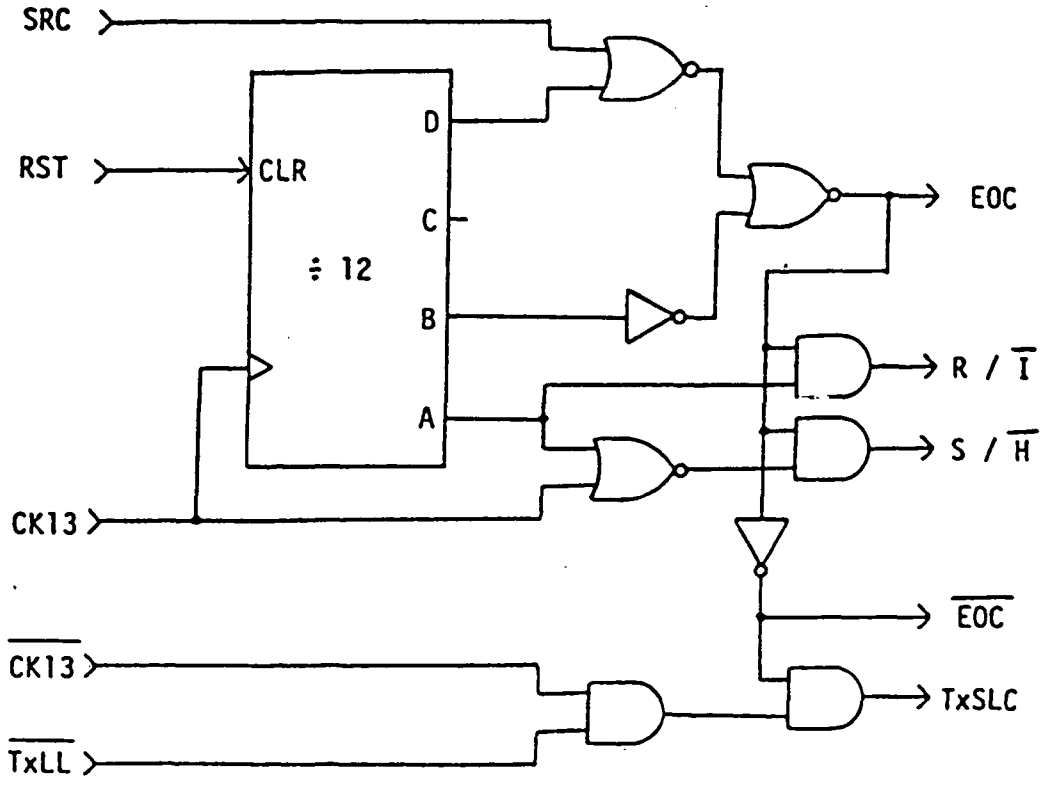


Figure 11. Cycle Controller for the IFC.

The circuit uses the bi-phase clock signal CK13 as a base clock for its operation. The divide-by-twelve counter is used to generate an overall cycle timing function for the IFC.

The cycle control circuit sends pulses to the Transmit Pulse Generator at the rate of CK13 or 30 KHz. The integrator will normally integrate ten radar return pulses to obtain each analog output data value. The circuit allows the processor to select either ten or four integration cycles per datum with the SRC, Sample Rate Control, signal. SRC originates from the general control register (Channel 2). The four integration cycle selection could prove useful if return signals are too large and saturate the integrator when ten cycles are used. In addition to these integrator cycles, two more cycles of CK13 are used for sampling the integrator output and resetting the integrator. When SRC is low, selecting ten integrator cycles (12 overall), the IFC's overall cycle rate is 2.5 KHz which corresponds to an analog sample available every 0.4 mS. SRC high selects four cycles (6 overall) for a cycle rate of 5 KHz or a 0.2 mS cycle time.

Figure 11 also presents a timing diagram for the cycle controller. Note that during integrator sampling and reset operations, the Transmit Shift Load Control (TxSLC) is inhibited to prevent radar output. TxSLC also may be inhibited by a low level on Transmit Load Lock (TxLL). TxLL is another signal from the general control register. The inverse of the End of Cycle (EOC) signal is used to enable the range gating function of the IFC. This way, range gating is inhibited during the integrator sampling and reset operations. The timing diagram also shows that the Sample/Hold control signal (S/H)

precedes the Integrator Reset Signal, (R/I), by half a cycle of CK13 or 16 microseconds. This gives the Sample and Hold device time to enter the HOLD mode. Note from the timing diagram that the number of integration cycles is unspecified for the first IFC cycle after a board reset (RST). The first EOC interrupting the processor after a reset should therefore be ignored. All cycles after the first are valid.

Programmable Transmit Pulse Generator

The IF transmitter requires a signal from the IFC to tell it when to transmit a radar pulse and for how long. The Transmit Pulse Generator does just that. Figure 12 is a block representation of the generator circuit. The 8-bit shift register in the circuit is parallel loaded with a pattern obtained from the latch of IFC Channel 3. The pattern is shifted out of the register serially to produce the transmit pulse. The shift register can be clocked at either 60 MHz or 30 MHz depending on which clock signal is jumper selected, CK0 or CK1. Normal configuration is 60 MHz. The circuit is told when to load and when to shift by the TxSLC (Transmit Shift Load Control) signal output by the Cycle Controller. The register is loaded when TxSLC is low and begins to shift after the rising edge.

The processor must load the Channel 3 latch with the proper pattern for the circuit to operate. By using a programmable shift register the generated transmit pulse can be adjusted to accommodate switching characteristics of the mixer used to produce the 60 MHz radar pulse. The processor has the ability to read the contents of the shift register through the buffer of Channel 3. The shift register can

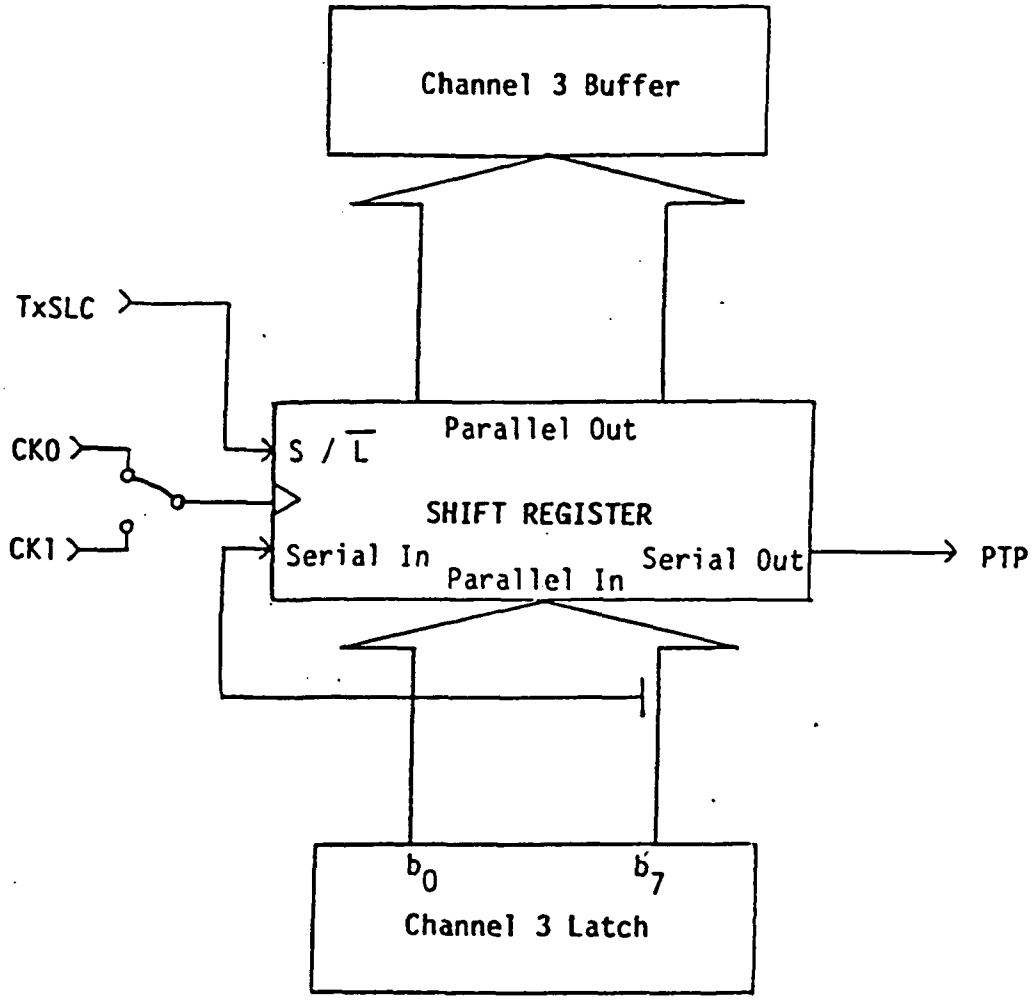


Figure 12. IFC Programmable Transmit Pulse Generator.

be read "on the fly" or if TxLL is low the loaded value can be read and confirmed. TxLL is set low through the general control latch (Channel 2).

Fixed Delay Generator

After generation of a transmit pulse, the IFC must wait a fixed amount of time corresponding to the minimal delay before the radar signal is returned to the input of the integrator. The Fixed Delay Generator accomplishes this function. The CK9, CK10, CK12 and CK13 clock signals are combined to produce this fixed delay output signal, FDS. Figure 13 shows the Fixed Delay Generator circuit and the timing of its output with respect to TxSLC.

The FDS signal is "OR"ed with Receiver Load Lock (RxLL) to produce the output signal Programmable Delay Count Load Control (PDCLC). PDCLC is applied to the next stage of the IFC, the Programmable Range Delay Generator. RxLL is one of the general control bits from IFC Channel 2's latch. This gives the processor a way of stopping the action of the next two stages of the IFC (Range Delay and Gate Pulse generation). The Fixed Delay Adjust signal (FDA) also originates from the control latch. This gives the processor a way of adjusting the fixed delay that is generated by the circuit. When FDA is low CK9 is included in the generator circuit and the fixed delay is approximately 13 microseconds. If FDA is high CK9 is ignored and the generated delay is approximately 11 microseconds. Note also that the two biphase CK13 clock signals can be switched by jumper selection to choose whether the falling or rising edge of CK13 starts the fixed delay. Fine

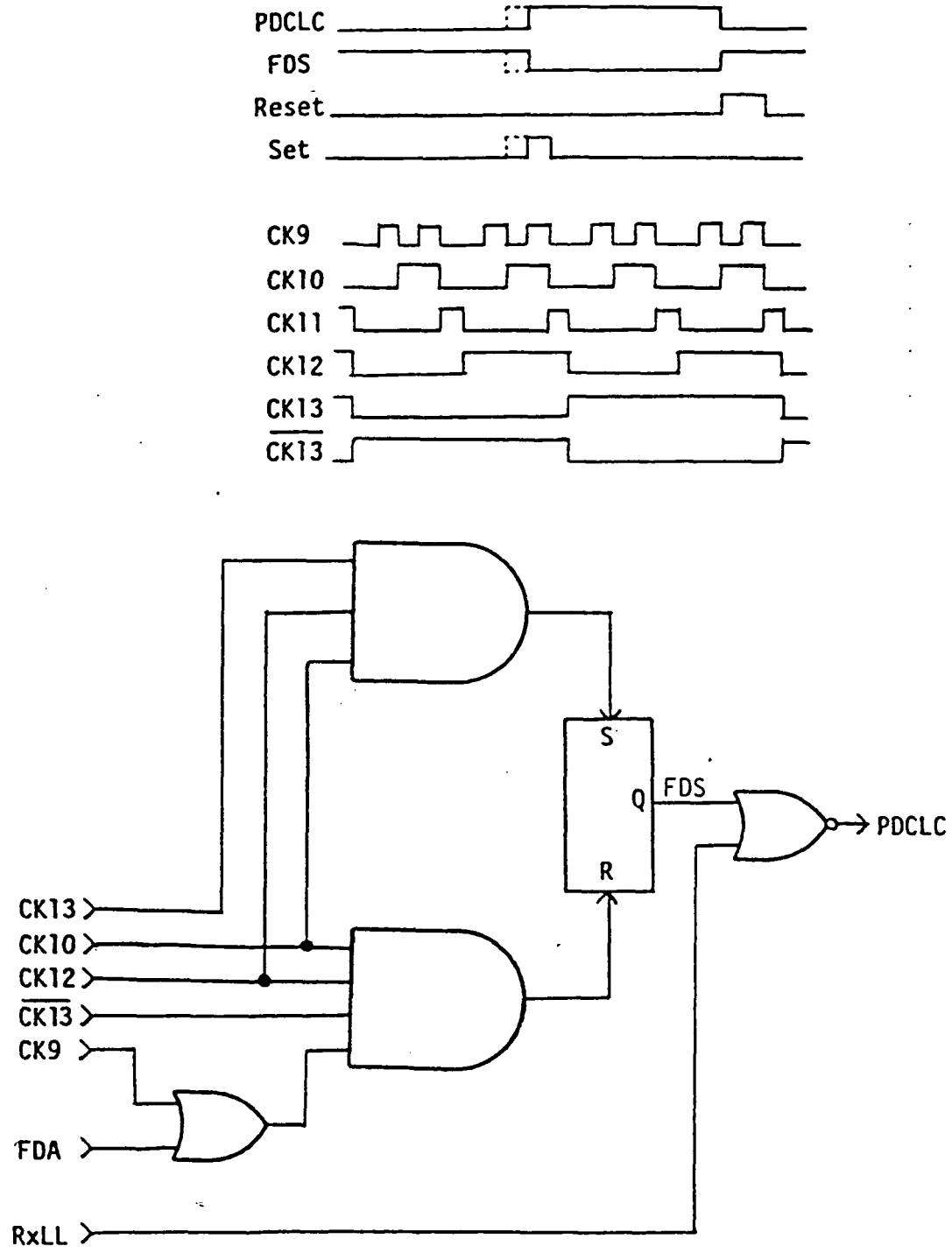


Figure 13. IFC Fixed Delay Generator.

adjustments are handled by the programmable feature of the Range Delay Generator.

Programmable Range Delay Generator

Operation of the radar system at varying elevation angles has two effects on the radar return signal in terms of range gating. The return signal delay varies with changes in distance to the target due to the electromagnetic waves propagation time. In addition, the width of the return radar pulse varies due to changes in the incidence angle of the wave on the target. The signal is spread through time as the target scene area increases. To accommodate for such changes and still provide a good noise rejection to the receiver, the range gate signal must be as narrow as possible and must occur at the proper time to "catch" all of the return signal. The Programmable Range Delay and Gate Pulse Generators allow the proccessor to adjust the range gating function for such changes.

The range delay generator is shown in Figure 14. This circuit uses a programmable counter that is loaded from the output by the latch of IFC Channel 4. The counter is clocked by CKO at 60 MHz. The F/F in the circuit catches the rising edge of the most significant bit out of the counter. The output of this F/F is the generated delay signal, PDS. By writing the proper value out to Channel 4 of the IFC, the processor can adjust the range delay that is generated. The generator's operation cycle is based on the signal provided by the Fixed Delay Generator, PDCLC (Prog. Delay Count Load Control). When PDCLC is low the counter is loaded and the F/F is cleared. After the rising

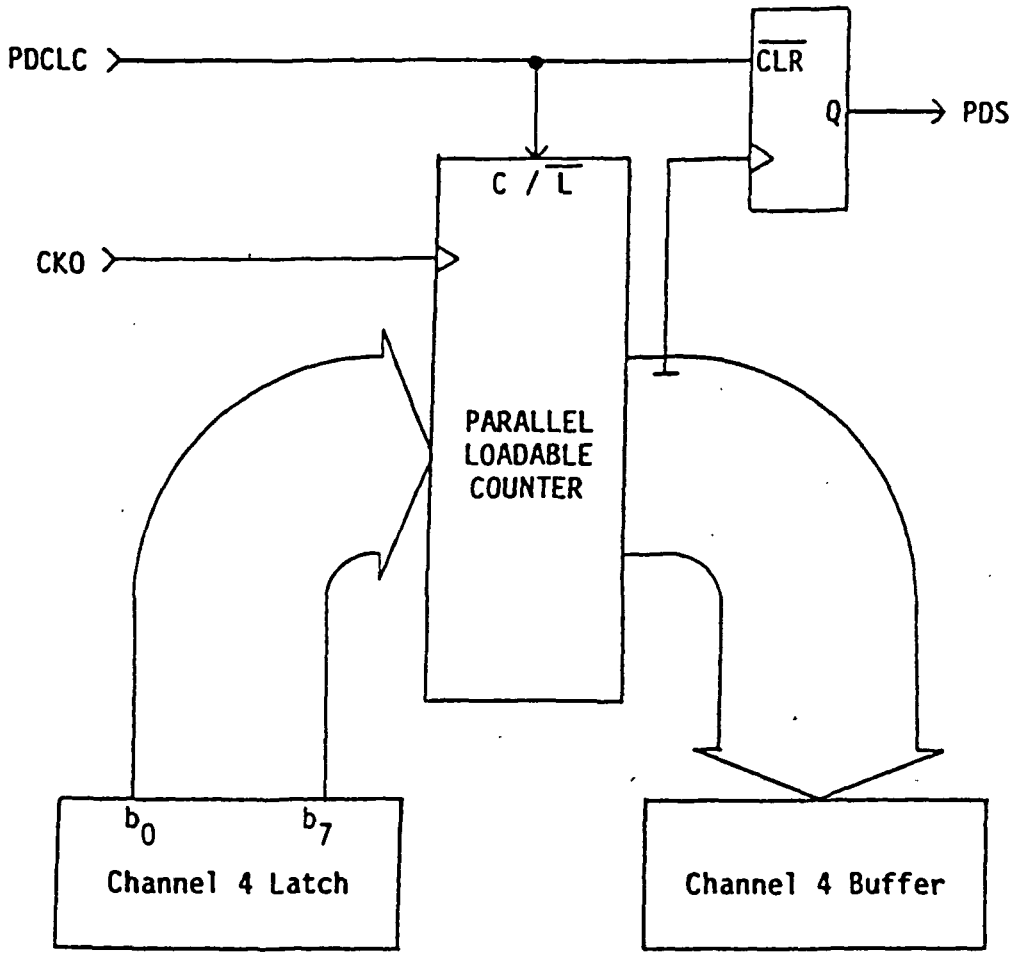


Figure 14. Progammable Range Delay Generator.

edge of PDCLC, the counter begins operation marking the start of the programmed delay. Note that the output of the counters can be read by the processor through the buffer of Channel 4. As with the Tx. Pulse Generator, the read is either "on the fly" or if RxLL is high the loaded value is read.

Programmable Gate Pulse Generator

The Programmable Gate Pulse Generator function is produced using the same circuit used for range delay. The only difference is that the circuit's function is controlled by PDS rather than PDCLC. PDS acts as the count load signal for the gate pulse generator. The rising edge of the Programmable Gate Signal (PGS) that is produced actually marks the end of the gate pulse. The rising edge of PDS marks the start of the gate pulse. By ANDing the PDS signal with the inverse of the PGS signal, the desired Programmable Gate Pulse signal, PGP, is generated. This signal is ANDed with EOC and inverted to produce the actual signal used to control the integrator gating circuit. Including EOC in the signal inhibits range gating while the integrator is being sampled and reset. Figure 15 presents the combined Range Delay and Gate Generator circuits and a timing diagram that illustrates how the circuits produce the gate pulse. The gate width is programmed and checked through IFC Channel 5. For reading the loaded value of the Gate Generator, the RxLL signal should be high.

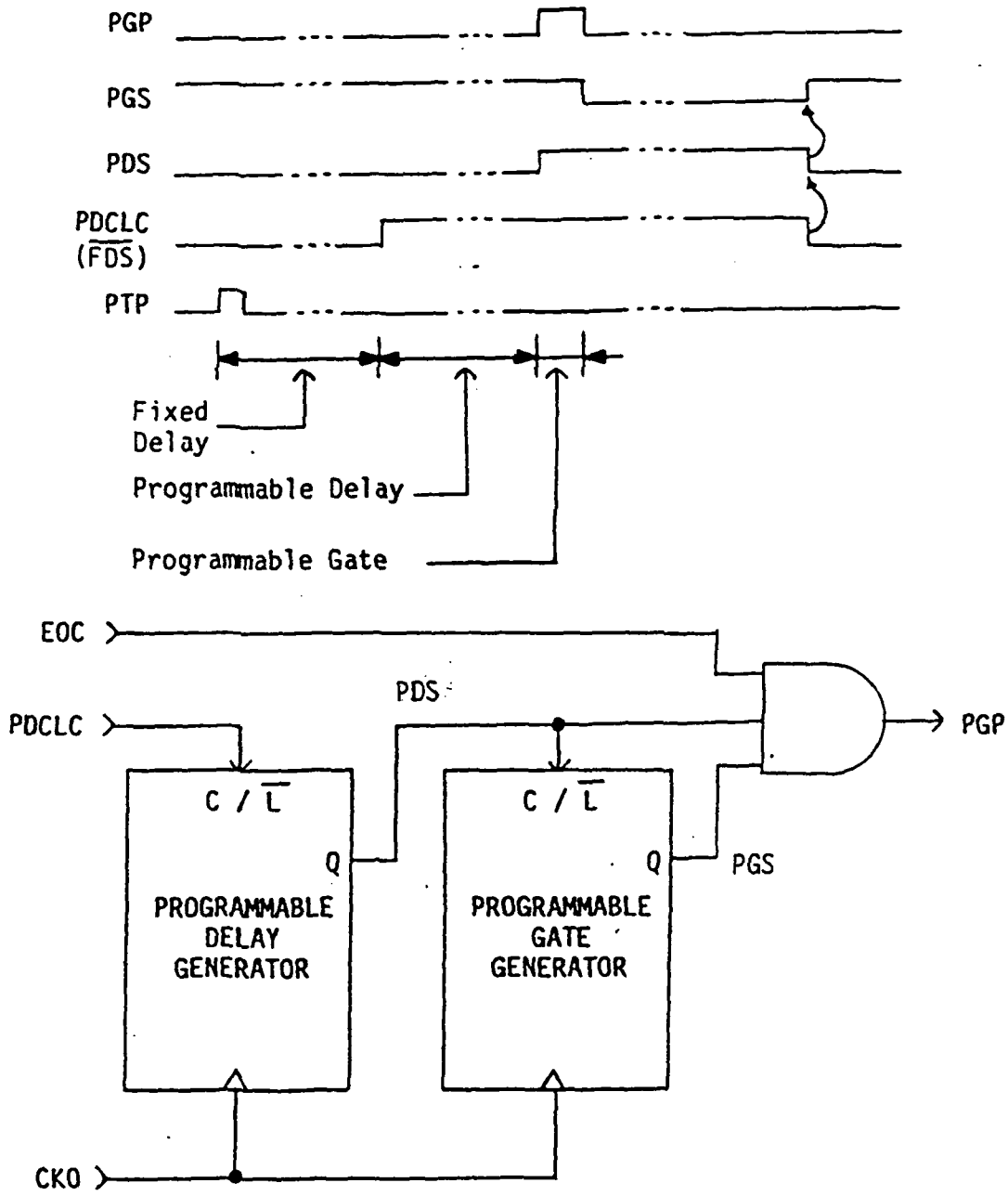


Figure 15. Programmable Gate Pulse Generator.

I/O Expansion in the IFC

The processor interface scheme used in the IFC provided an easy way to expand the processor's I/O capability in the IF section. Two additional IFC Channels are used to accomplish this expansion. Channel 6 is used for controlling the Digital Attenuator in the IF receiver. The output of Channel 6's latch is returned through the buffer of that channel to allow the processor a method of checking the attenuator's setting. Channel 7 is used to control and monitor the RF Head Multiplexer in the IF section. Switches identical to those used in the RF polarization control are used for the multiplexing function in the IF section. Monitoring of the multiplexer switches is accomplished the same way that monitoring of RF polarization switches was done.

A complete set of detailed circuit diagrams for the IF Controller circuit board can be found in Appendix A. The IFC was first prototype on a special Schottky TTL wire wrap board. Once made operational, the final circuit was produced in the form of a printed circuit board. A print of the circuit layout can also be found in Appendix A along with a component placement diagram.

When combined, the RDADS Processor and the IF Controller provided all the hardware needed to control and operate the RPS. Since most of the hardware functions are under software control, the operation of the RPS can easily be adjusted or modified. The next section presents the software developed for the RDADS.

SOFTWARE

RDADS hardware was selected and designed to permit flexibility in the system's implementation. This flexibilty was achieved through a microprocessor based system design. The polarimeter's functional operation was defined primarily through the microprocessor software. This section presents the design and development of the RDADS software.

The RDADS is a complex processing system that performs a number of functions during operation. The software design problem had to be divided into smaller problems and organized for development. A structured programming approach was taken for the design and development of RDADS software. This section will begin by briefly presenting the programming techniques that were chosen and how they fulfill the project requirements. Emphasis and elaboration will follow in a discussion of the actual software.

Structured Programming

Appearance of a computer program, its readability, comprehensibility, and "style", is the principal aspect of "structured programming" [14]. Structured programs are readable, easily understood, and well organized. Modular programming and limitation of program constructs are two important concepts used in developing programs that are structured.

Modular Programming

Modular design and implementation breaks the programming problem down into levels and blocks. Levels refer to hierarchical divisions within an installation. At each level the entire installation can be divided into blocks with independent characteristics. Blocks and their hierarchical levels are commonly illustrated by block structure diagrams. An example of a structure diagram is shown in Figure 16. Here the entire problem is represented by Block A at Level I and is broken down into Blocks B, C, and D at Level II. The problem breakdown can be carried through as many levels as required. This modular breakdown allows the problem and solution to be viewed in terms of simpler components and in an organized fashion.

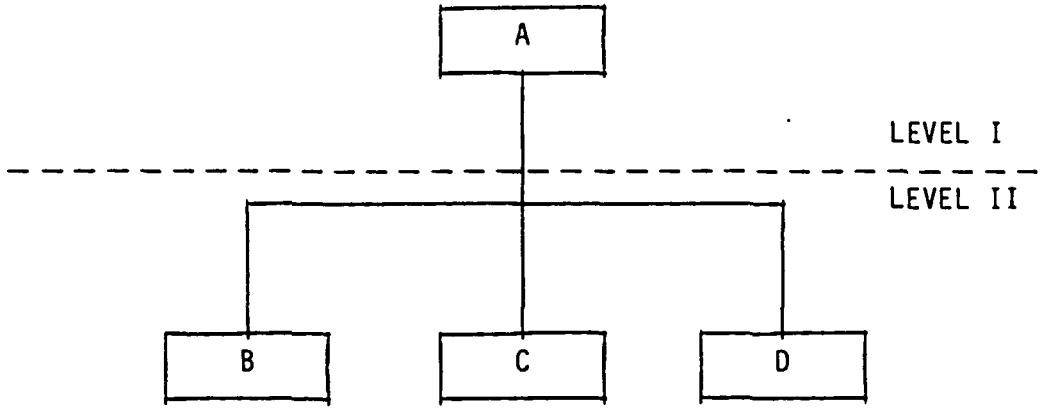


Figure 16. Simple Block Structure Diagram.

Limited Constructs

A second requirement for structured programming is the concept of limiting the number of basic program constructs [14, 15, 16, 17]. A program construct can be defined as a structure element or pattern used to express logic in a program. There are three basic constructs

necessary to fulfill all programming requirements. These basic constructs, referred to as sequence, selection, and iteration, can be combined in any fashion to accomplish any logical task [18].

The construct of sequence is a succession of selection structures, iteration structures, and individual process steps. To express sequence, statements are written one after the other, as follows:

```
Input X and Y
Add X and Y
Output result
```

Selection is a construct in which a process is broken into options. The execution of these options is dependent on a specified condition. To illustrate selection, terms like IF, THEN, and ELSE are commonly used, as in:

```
IF  X > Y  THEN  MAXVAL is X
           ELSE  MAXVAL is Y
```

Repetition of a process is accomplished using the third construct, iteration. A DO loop is the common example of iteration such as:

```
DO I from 1 to 10
Add X(I) to SUM
End DO
```

With these basic programming patterns any logical process can be implemented. Processes more complex than those illustrated above are effected by combining these basic constructs.

Development Support

Both of the above conceptual components are fundamental to the

structured programming philosophy. Many available software development tools and programming languages directly support the design and implementation of structured programs through the use of these concepts. Programming languages usually provide a means of directly implementing the three basic constructs required. This allows the design portion of the development process to be virtually language independent. A software design can be entirely depicted in pseudo-code which is an informal method of expressing the steps of a process in terms of the basic constructs. Since these constructs are supported directly in most languages, the implementation phase is accomplished through simple conversion of pseudo-code to actual source code. A programming language can be chosen at the time of implementation without affecting the logical design of the program.

Modularity is promoted not only in languages but also by support tools such as program library facilities, module linkers, and pre-developed modular process utilities. Most all languages directly support modularity by providing for some type of subprogram, function, or procedure implementation. Once developed, program modules can be stored as individual units or as part of a library of modules. An entire software installation is obtained by combining individual modules together through the use of an object linker. The linker also provides the function of "looking-up" and extracting needed modules from libraries. Commercially available process utilities can be also linked with user modules to provide the complete system installation. These and other software design tools and products support and simplify the development of structured programs.

Software Design

In this section a statement of the software problem will be given. Some of the specific requirements on the software installation will be itemized. This will be followed by a general description of the design approach taken. Refinement of the design problem and descriptions of solution implementations will be given later in this section.

Problem Statement

RDADS software was required to perform three basic functions. These were:

- o Communicate with the main computer in the data van,
- o Control radar operation and data acquisition,
- o Position the RPS antennas.

The first two functions are explained here. Antenna positioning software was developed external to this project.

Radar Control and Data Acquisition

Radar operations can be divided into three categories. Two of these are the microwave transceiver control operations and the IF section control operations. The third is the acquisition of data from the radar.

Transceiver operation is controlled with RF switches. There are three RF switches under computer control in each microwave transceiver. These are the CAL/OP switch and the transmit and receive

polarization transfer switches. All microwave switches are controlled and monitored through parallel I/O operations from the processor.

IF section operations include setting and checking the control and parametric registers of the IF Controller. IF Controller registers include a general control/status register, a transmit pulse register, the range and gate width registers, a digital attenuator register, and a transceiver multiplexer register. IFC communications are also performed through parallel I/O by the processor.

Radar data acquisition is accomplished primarily through conversion of the analog signal output by the IF section. The IF Controller board signals the computer when the analog signal is ready for conversion. The computer then converts the analog signal to a digital count. It can determine from this measurement if the Digital Attenuator should be adjusted. Control of the attenuator is performed automatically by the processor to implement a Digital Automatic Gain Control (DAGC) function for the receiver.

RDADS Communication

Operation of the RDADS is normally controlled by the data van CPU. The RDADS can also be controlled directly by the user. RDADS communication was designed to allow for two different operational set-ups. The first is the normal set-up of communicating with the data van in a remote fashion. The second is a local set-up using a console terminal in place of the data van CPU. Using a terminal to communicate with the RDADS allows the system to operate in a stand alone configuration instead of being a slave device to the main CPU. This local set-up was

devised to provide for testing, maintenance, and calibration. Communication software was designed not only for the basic operational requirements, but also for flexibility and utility in testing and maintaining the RPS.

Design Approach

The approach to the design of the software installation was to develop an operating system that allowed interactive control of predefined functions. Interaction was to be either directly through a console terminal or indirectly through the data van CPU. A general command list was defined that would provide for all the required operations. Each command task was broken down into steps needed in accomplishing the overall function. By breaking each level of tasks down into a subordinate level, the actual operational requirements of each task were divided into simpler components. This breakdown continued until the sub-divisions of a task reached a point where they could be implemented easily.

The operating system for the RDADS is called the Remote Operations Utility (ROU). Figure 17 presents a task structure diagram of the ROU. Each task performs a command function except for the Radar Task which is a free running task used to acquire the radar measurement data. Each of the command tasks will be presented here by describing the format of the command entries and the actions they perform. This will be followed by a discussion of the radar task's logical operation.

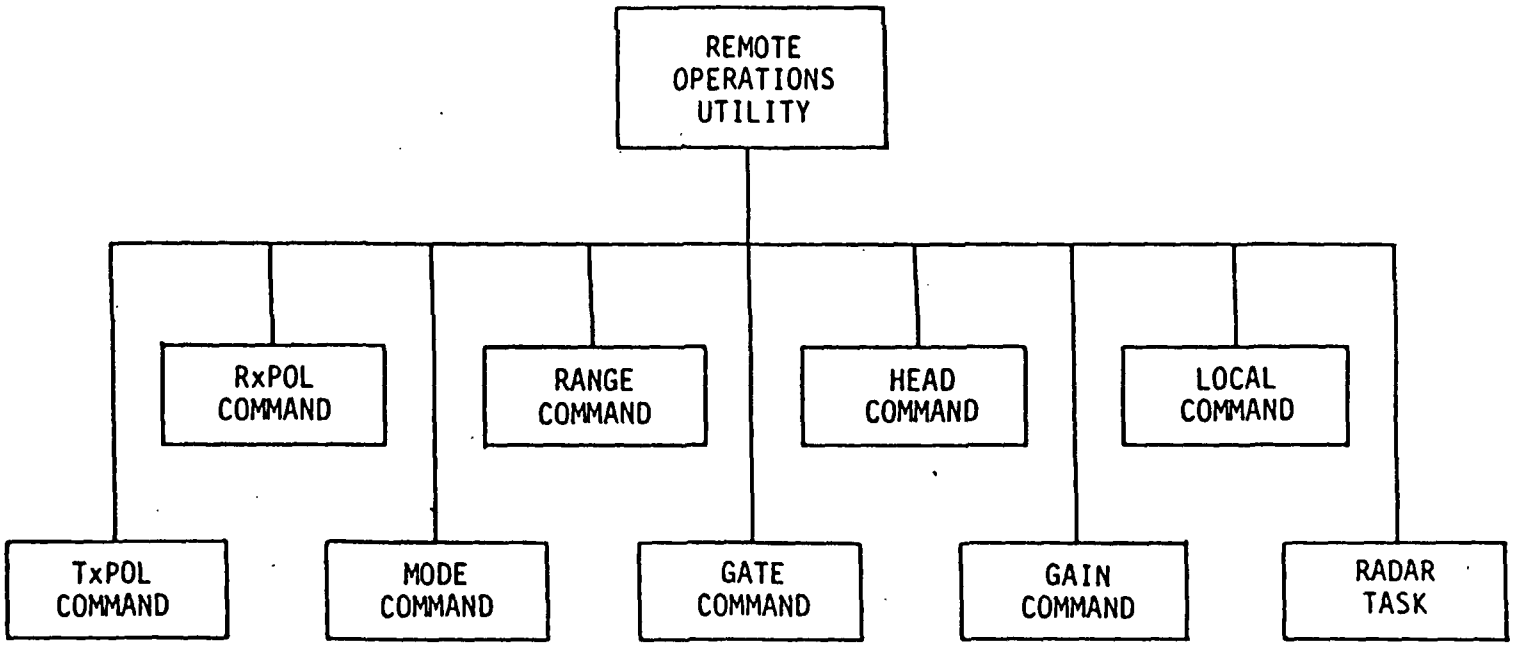


Figure 17. Structure Diagram for the Remote Operations Utility Package.

ROU Commands

The following rules apply to all the descriptions of the Remote Operations Utility commands:

1. Keywords are shown in upper case and must be entered as shown. Any abbreviation of the keyword is acceptable but should contain at least enough characters to distinguish it from any other keyword used in the command. Command keywords may be abbreviated but also must contain enough characters to distinguish them from other commands. Failure to do this may cause unwanted results. Ambiguity among keyword entries is resolved by order of appearance in the format description. Ambiguity among command keywords is resolved by order of appearance in the command table. Abbreviations are formed by entering the first characters of a keyword in the same order that they appear in the keyword. Any number of characters is acceptable as long as no intermediate characters are omitted.
2. Command keywords are always the first keyword in a command format. Everything that follows the command keyword is referred to as the command tail. The command keyword and its command tail constitute a command string or command entry.
3. All keywords, symbols, and variable entries must be separated from other keywords, symbols, and variable entries with one or more spaces.

4. Braces, " { } " group alternatives in which one and only one alternative must be chosen.
5. Parenthesis, " ( ) " group alternatives in which one or more must be chosen. At least one alternative must be chosen but two or more can be used.
6. Brackets, " [ ] " group options which may be omitted entirely. Underlined options are assumed if all options of the group are omitted.
7. The "or" symbol, " | " separates command options or alternatives.
8. Braces, parenthesis, brackets, and the "or" symbol are used to identify options or alternatives only and are not intended to be entered as part of the command string during operation.
9. Rules indicated by symbols that enclose groups apply only at the level of the group they are enclosing. When an option or alternative is a group, the rules of the outer group apply to the nested group as a unit. Within a nested group the rules of that group apply.
10. Lower case letters are used as symbols for variable entries. Permissible values or ranges for the variable are specified following the command's format description. A value for the variable must be entered unless the entire option in which it appears is omitted.
11. Several commands have a query option that is selected either by following the command keyword with a question

mark, " ? ", or by default. If the command keyword is entered by itself or followed by a question mark it is interpreted as a query only and has no effect on the current setting of the hardware or software entity that the command controls. The result of a query is simply a response of information on the current setting or status of the related entity.

LOCAL Command

RDADS remote communications were designed to be performed with either the data van CPU or a console terminal. In keeping with this approach it was decided to use two modes of operation for RDADS output. LOCAL mode designates that the RDADS processor is to communicate with a console terminal. REMOTE mode is used for communications with the main CPU. The only difference in operation for these two modes is the format of output data. In LOCAL mode the ROU provides processes for the screen formatting of data displayed on the console. This formatting was intended to aid in testing, maintenance and calibration. In the REMOTE mode of operation data are sent with a unique two character ASCII introducer string that identifies the data. The data follows this string in a preset format and is terminated with a standard sequence consisting of a carriage return and line feed. To select between the two modes of operation the LOCAL command is used. The format of the LOCAL command is as follows:

LOCAL [OFF | ON]

The user enters the command name followed by the desired setting of either ON or OFF. If the selection is omitted, ON is assumed. By default, at system start-up time, the ROU is in the local-off or REMOTE mode. When local-on is specified using the LOCAL command the ROU outputs control sequences that initialize the console terminal screen. The console is divided into two sections, one for display of system parameters and status and one for command entry and responses. When the LOCAL command is used to set the communications mode a mode flag (SDL$MODE) is set to identify the currently selected mode. The flag is set to either REMOTE (literally OFFH) or LOCAL (literally 0). This flag is then used to control the output of responses to command queries. In the LOCAL mode, response to any command query usually appears in the section of the screen reserved for parametric and status information. When mode is set to REMOTE (local-off), query responses are sent with an ASCII introducer followed by the desired information and a standard termination sequence.

TXPOL and RXPOL Commands

Two commands, TXPOL and RXPOL, control and monitor the polarization settings of the microwave transceivers. Both commands are identical in format and operation. Transmit polarization for all of the microwave transceiver heads is controlled and checked with the TXPOL command. The RXPOL command performs this function for receive polarization. Polarization settings for both transmit and receive are set the same for all microwave heads. This has no impact on operation since only one head is in operation at any given time. The format for

these two commands is:

{ TXPOL | RXPOL } [= {HORIZONTAL|VERTICAL} | ?]

To set the transmit or receive polarization, the proper command keyword is entered followed by an equal sign, "=" and the desired setting specified as either HORIZONTAL or VERTICAL. The current setting of the transmit or receive polarization can be checked by entering the command keyword alone or followed by a question mark. The response to a query when in the REMOTE mode would be introduced by the letters "TP" for transmit polarization or "RP" for receive polarization. This is followed by a character that indicates the current status, either an "H" for horizontal or a "V" for vertical. The response is terminated by the standard sequence. When in LOCAL mode the response is written to a reserved location on the console screen.

MODE Command

To provide for desired operational capabilities, four different modes of operation were designed for the radar system. These four mode settings were actually four different combinations of two mode characteristics. Each characteristic could assume two possible states. The first mode characteristic refers to the physical operational setting of the microwave transceiver. Each head can be set into either a CALIBRATE or OPERATE mode. CALIBRATE is used for taking an internal calibration measurement through the transceiver. OPERATE provides for setting the transceiver in a mode that enables transmission and reception through the transceiver's antennas. Actual target scene measurements are accomplished in the OPERATE mode. A second radar mode character-

istic involves the repetition of measurement acquisition. The two settings for this were SINGLE and CONTINUOUS. To acquire one measurement return value the radar was placed in SINGLE mode. CONTINUOUS mode provided for multiple radar measurement acquisitions. When in CONTINUOUS mode the radar measurements are made continually until the mode is reverted back to SINGLE. The MODE command was developed for selecting the various radar modes of operation. Its format is as follows:

MODE [= ({CALIBRATE|OPERATE} | {SINGLE|CONTINUOUS}) | ?]

In order to specify or change the mode settings of the radar system, the command keyword MODE is entered, followed by an equal sign. After the equal sign either or both of the desired modal characteristics can be entered. As an example the radar could be set to OPERATE-CONTINUOUS with the following command string:

MODE = OPERATE CONTINUOUS

The mode could then be set to OPERATE-SINGLE by entering the following:

MODE = SINGLE

Since the mode was already set to OPERATE, only SINGLE had to be specified. To change to CALIBRATE-SINGLE the command string that follows could be used:

MODE = CA

Note that CALIBRATE was abreviated here to CA. "C" alone would be ambiguous since CONTINUOUS also begins with the same letter, so "CA" was the minimal abbreviation.

The effect of the MODE command is to set the currently selected transceiver (see HEAD command below) into the specified calibrate or operate mode. All other heads are set into calibrate to effectively

turn off those transmitters. In addition the range delay and gate width parametric registers in the IF Controller are set to the current values saved for the specified mode (see RANGE and GATE commands below). The effect of specifying SINGLE or CONTINUOUS in the MODE command is to set the radar task into that particular mode of operation. How this affects the radar operation will be discussed in the description of the radar task.

To check the current mode setting, the MODE query command can be used. This is simply the command keyword alone or followed by a question mark. The response to a MODE query in the LOCAL mode would be to print the word CALIBRATE or OPERATE followed by the word SINGLE or CONTINUOUS in a reserved location on the console screen. In the REMOTE mode the response would be "MO" followed by two letter characters to indicate the current mode combination. The first letter would be either a "C" or "O" to indicate CALIBRATE or OPERATE respectively. The second letter would be either an "S" or "C" to indicate SINGLE or CONTINUOUS respectively. It is up to the user to know the first "C" would refer to CALIBRATE and the second would refer to CONTINUOUS. The remote response is terminated with a carriage return and line feed.

HEAD or BAND Command

Selection of a microwave tranceiver head was provided for with the HEAD or BAND command. This is one command with two valid command keywords. Either keyword can be used with the same effect. Both were provided for clarity to the user. Original testing was done with transceiver interconnections assigned a number 1 through 4 for handling

up to four microwave heads. The actual frequency band associated with each of these head numbers was not assigned until the final installation of the transceivers. Each band is referred to by a band letter "X", "L" or "C" which were assigned to the head interconnects "1", "2" and "3" respectively. Interconnect "4" was assigned the letter "R" for a reserved band designation. The command string for selecting the transceiver to be used is entered with either command keyword, HEAD or BAND, followed by the intended selection specified using either the numeric or letter designation assigned to the desired head. The command format is:

$$\{ \text{HEAD} \mid \text{BAND} \} [= k \mid \underline{?}]$$

where k is one of the numerals 1, 2, 3, or 4 or one of the letters X, L, C, or R. There are two effects of this command when a transceiver selection is specified. First the transceiver multiplexer is set to select the specified transceiver. Secondly, the selected transceiver is placed in either the calibrate or operate mode as specified by the current mode setting for the radar. All other transceivers are placed in the calibrate mode to disable transmission from those bands. The command will also respond to a query with the current setting of the command's entity. In the REMOTE mode this response is "HE" followed by the head number that indicates the current setting. This is followed by the standard termination sequence. In the LOCAL mode the number and letter are both displayed in reserved locations on the console screen labeled "HEAD = " and "BAND = " respectively.

RANGE and GATE Commands

Setting and checking the range delay and gate width parameters of the IF Controller board were provided for with two commands, RANGE and GATE. Each of these commands has associated with it two software parameter entities. One is for a calibration value of the parameter and the other is for an operation value of the parameter. There are then four values in all that are saved internally: Calibrate Range Delay, Operate Range Delay, Calibrate Gate Width, and Operate Gate Width. The current radar mode setting determines which range and gate values are used in the IFC's parametric registers. When the mode is changed, the range and gate parameteric registers are automatically set to the proper value as determined by the current value saved for the associated parameter. The RANGE and GATE commands have basically the same format. It is as follows:

{ RANGE | GATE } [[CALIBRATE|OPERATE] = n | ?]

Here "n" should be replaced with an integer value ranging from 0 to 255. This value is the number of 16.67 nS time divisions that the specified parameter is to be set to. To specify the setting for a range delay or gate width, the proper command keyword is entered first. This is followed by a keyword to identify that either the calibrate or operate value is being set. If the second keyword is omitted OPERATE is assumed. Next an equal sign should appear in the entry followed by the integer value desired. The effect is to store this value in the proper parameter variable as specified. If the value is specified for the current mode of operation then the command also

sets the associated IFC parametric register to the specified value.

The current settings of both the calibrate and operate values for the range delay or gate width parameters can be obtained from the system using the query version of the RANGE and GATE commands. For either command the response to a query in LOCAL mode is to write both the calibrate and operate values for the specified parameter in a reserved location on the console screen. The value that corresponds to the current setting of the related IFC parametric register is underlined for emphasis. In the REMOTE mode the response will begin with an introducer of either 'RD' for range delay or 'GW' for gate width. This is followed by the calibrate value. A semicolon, ";" is then added as a delimiter followed by the operate value also. Both of the response values are sent as ASCII encoded hex values that range from 0 to 0FFH. As always the REMOTE response has the standard terminator.

IFGAIN Command

The IFGAIN command was devised to provide the capability to set the IF receiver gain manually. This command allows the operator to program the gain setting of the IF receiver directly without using the automatic gain control feature of the radar task. IFGAIN has its primary use in system testing and adjustment. For any practical use the radar must be in single mode of operation before the IFGAIN command is used. Since the radar task changes the gain for data acquisition, the gain setting entered using the IFGAIN command is valid only while the radar task is disabled (ie. SINGLE mode). The format of the IFGAIN command is:

IFGAIN [= i | ?]

where "i" is replaced by an integer value ranging from 0 to 127. The actual gain is the integer value multiplied by 1/2 dB. When the gain value is specified using the IFGAIN command, the value is directed to the IFC where it is used to set the digital attenuator. This attenuator as discussed in the previous section is a 7 bit digital attenuator that has settings ranging from 0 to 63.5 dB in 1/2 dB steps.

In response to the IFGAIN command query while in LOCAL mode the gain setting of the IF receiver is displayed at a reserved location on the console screen. When in REMOTE mode the response is "GN" followed by the gain setting in ASCII encoded hex (0 to 7FH) and a carriage return/line feed termination.

Radar Task

The radar data acquisition task is covered in this section. A description of the task is given explaining its logical execution. The primary aspect of the radar task, the Digital Automatic Gain control function, is covered in detail.

Radar Task Operation

Radar data acquisition, the primary function of the RPS, is handled by a stand alone free running task. This task, the radar task, falls under the control of the operator via the MODE command. It can be enabled or disabled by setting the radar mode of operation properly. Once enabled, the task runs a data acquisition process to obtain a radar data value. A return data value is obtained for each pass or iteration of the task's data acquisition process loop. The task can be

enabled for either a single pass or for continual iterative passes on this loop. This equates to obtaining either a single radar return value or obtaining return values continuously. Once in the continuous mode, data acquisition can be halted by returning to the single mode. In single mode the radar task acquires a single data value and disables itself. It remains disabled until it is re-enabled by the MODE command task. Thus for single mode the radar is in operation only long enough to acquire one data point and then it returns to an idle state.

The logical execution flow of the Radar Task is presented by the descriptive pseudo-code that follows:

- o Initialize the Radar Task's software data structures and necessary hardware.
- o Begin a DO-FOREVER loop.
  - o Get mode specification from the MODE command.
  - o Do while mode is SINGLE or CONTINUOUS:
    - o If mode is SINGLE set to OFF.
    - o Obtain radar data value.
    - o Display radar data value.
    - o Get new Radar Mode if available from MODE command.
  - o End of Do-while.
- o End of Do-forever.

This task has the general task format. It begins with an initialization section that is executed once. This is followed by the operation section which is a loop that is executed indefinitely.

The first step of the operation section is to obtain the radar

task mode that is provided by the MODE command. This mode is either

- o Display radar data value.
  - o Get new Radar Mode if available from MODE command.
  - o End of Do-while.
- o End of Do-forever.

This task has the general task format. It begins with an initialization section that is executed once. This is followed by the operation section which is a loop that is executed indefinitely.

The first step of the operation section is to obtain the radar task mode that is provided by the MODE command. This mode is either SINGLE, CONTINUOUS, or OFF. Execution of the next step of the operation loop is based on this mode setting. When the mode is OFF the task does nothing for the remainder of the loop. When the loop is repeated it will execute based on whatever the mode setting is at that time. If the mode is SINGLE or CONTINUOUS the task operates to obtain radar data.

There are several steps involved in obtaining a radar data value. As explained in the IF Hardware Description section, the IF receiver of the radar has a computer controlled digital attenuator that is used to vary the receiver's gain. The receiver provides an analog DC voltage level output from a sample and hold device to the RDADS processor's analog converter that is proportional to the radar return energy. The converter digitizes the analog signal to obtain an integer value representative of the signal's voltage level. In order to operate the

radar, it is necessary to control the IF gain to maintain the receiver's operation in a linear region and keep it out of saturation. The basic concept is to maintain the output voltage within a window that is known to be a linear region of operation for the receiver. To do this the analog output is monitored by the computer and, based on the signal's level, the gain is controlled via the digital attenuator. This is an iterative process in which, based on the signal level returned for the previous gain setting, the gain is changed until the signal level is driven to the desired operational limit. The steps needed to accomplish one iteration of this gain setting process would be to first set the gain to some prescribed value, that is to either an initial value or a value based on resulting return level of the previous settings. Next the return level for this new gain setting must be measured through the use of the analog converter. Finally the value obtained from this measurement is compared to preset values to determine how the gain should be adjusted for the next iteration. Until the gain is set to a value that forces the return voltage level into the desired window, the gain changing process is repeated. This technique of controlling the gain is a Digital Automatic Gain Control (DAGC) function. The particular method used to accomplish the DAGC function will be discussed in the next section.

Digital Automatic Gain Control Implementation

The DAGC performs a dual purpose. The function not only controls the gain of the IF reciever but it also provides the radar return data. The DAGC function was designed as a procedure that executes a controlling process each time it is called. The function returns, as a

procedure output, the gain setting that the control process generated. This gain setting is recorded as a single radar data point.

Recall that the digital attenuator effects a gain control on the receiver of 63.5 dB in 0.5 dB steps. This is accomplished by switching stages of various attenuation in or out of the input line. There are seven stages of fixed attenuation (32, 16, 8, 4, 2, 1, and 0.5 dB). The window chosen as the target for the IF receiver's operation was a narrow band centered in the middle of the negative voltage dynamic range of the analog input board. It was assumed that a change in gain of 0.5 dB at this operational point would effect a change of 0.25 volts in output from the receiver. This corresponded to an analog output voltage from the IF receiver between -2.375 volts and -2.625 volts. It was determined that this range would correspond to linear operation for all of the analog circuitry in the receiver section. The negative voltage output from the receiver section increases in absolute magnitude for increases in gain setting for a constant load on the microwave transceiver. To clarify the discussion absolute magnitudes will be used. Recall that the output values of the radar receiver are always negative. The window of concern was therefore between 2.4 and 2.6 volts in absolute magnitude.

It was initially attempted to implement the DAGC process as a binary search through the allowable differential gain settings to find the one setting that placed the receiver output into this window. In the binary search technique the gain was initially set to half of its maximum value (highest order bit high and all others low). The analog return was then compared to the operation window. If the voltage was

found to be of lesser magnitude than 2.4 volts, it would indicate that the gain had to be increased. If it were of greater magnitude than 2.6 volts then a gain decrease was in order. To increase the gain the high order bit remains high and the next lower bit is also set high. To decrease the gain the high order bit is reset to low and the next lower bit is set high. This corresponds to a change in gain of either plus or minus one half of the previous gain adjustment. With this new gain setting the entire process is repeated to determine if the gain is high or low for the desired operation. This bit-wise gain setting process is continued until all seven bits of the gain are set to the proper state that places the receiver's output into the desired range. Ideally this approach would be the most efficient since it takes only seven iterations for any possible gain setting. This approach worked, but only in certain cases. It was discovered that intermediate gain settings for relatively large radar return signals caused the video amplifier of the receiver section to "shut-down" which produced ambiguous analog input signals. These ambiguous signals completely invalidated the DAGC binary search control scheme. Another approach had to be developed.

A change of 1/2 dB in gain was experimentally determined to effect a change of at most 0.25 volts when the receiver was operated in the linear region. A new DAGC process was developed that made use of this fact. The process started by setting the receiver gain to a minimum. The gain is then increased in 4 dB steps until the receiver output is above 0.4 volts. The gain is next increased by steps of 1/2 dB until the output is above 2.4 volts. The output is at this point in the

desired window of operation so the gain setting is returned. This approach assumed that the maximum increase in output for a change in gain of 4 dB would be no more than 2.0 volts when the receiver output is below 0.4 volts. It also assumes that a 1/2 dB increase in gain will not increase the output more than 0.25 volts when the output is less than 2.4 volts in magnitude. As long as these two assumptions are correct the DAGC function should control the receiver gain within the 1/2 dB gain resolution to maintain the output as close as possible to 2.5 volts.

Real-Time Multi-Tasking

A real-time system is one in which events and processes take place in an asynchronous fashion. This means that events are not scheduled and occur at any time. These asynchronous events must be handled as the need arises, that is in a "real-time" fashion. Systems such as this are common in meeting the requirements of a physical world application. A real-time system must be flexible, changing its activities in response to unpredictable changes in its physical environment. When system processes handle asynchronous events, it is often necessary to be able to accommodate several events at once. In order to do this, it is required that multiple processes be able to take place at the same time. With regards to software, processes are sometimes called tasks and software systems that execute multiple processes concurrently are referred to as multi-tasking systems. The radar system is a real-time physical system with multiple events occurring asynchronously, requiring multi-tasking service. In the design approach given for the radar operating software, the installation consists of several command

tasks and a free-running radar task. This approach fits well into the framework of a multi-tasking system.

It is physically impossible to have more than one program or task running at any given time in a hardware system that has a single processor. It is, however, possible for the computer to share its resources with many tasks in such a way as to make it seem to the outside world that the tasks are running concurrently. To accomplish this there are three important requirements [19]:

1. There must be a means of notifying an executing system when important events occur in the physical environment.
2. The computer must accommodate more than one program in its memory simultaneously.
3. There must be provisions for stopping the execution of one program and starting the execution of another, and it is necessary to do this quickly and at any time. It must also be possible to resume a program at the point were it was stopped.

Less obvious is the fact that activities of one program often depend upon the activities of another. A means of communicating between programs must be provided to allow for these dependencies. Also, some programs can be more important than others, so a program priority scheme is required. Higher priority programs, however, must stop running to allow lower priority programs time to accomplish their task. Finally, it is desirable that an executing system make the fullest use of the processor's time. Typically, in a system where program scheduling is non-flexible, there are time periods when the processor is not

doing useful work (i.e. during I/O operations). Allowing other programs to run during these periods improves the processor's efficiency.

A software operating system that provides for these requirements is referred to as a Real-Time, Multi-Tasking Executive. There are several commercially available Real-Time, Multi-Tasking Executive packages sold to meet the typical needs of a real-time system. These "off-the-shelf" software pakages will usually provide many additional utilities that may or may not be required for any given application. Such utilities are provided to ease the program development job. By providing commonly used utilities as part of the package, program development efforts can be more effectively directed toward meeting requirements of the specific application. For these reasons the use of an "off-the-shelf" executive software package is extremely advantageous. One such package made available for use on this project was RMX-80, a product of Intel Corporation.

The RMX-80 Executive Package

RMX-80 provided many useful functions and utilities that made the program development process less cumbersome. Fundamentally, the executive provided for multiple independent tasks which compete with each other for system resources including CPU time. Three basic task interactions were accommodated with the use of RMX-80. Specifically these were:

1. Communication - The transfer of information from one task to another.

2. Synchronization - Executional sychronization of a task's activity with an external event, with the activity of another task, or with system timing.
3. Resource Management - Mutual exclusion of access to a hardware or software resource in order to prevent concurrent or disruptive use of the resource.

Tasks interact in RMX-80 through the use of exchanges which are data structures that are functionally a "mailbox" through which information is sent and recieved by tasks. This information is transferred in the form of a second data structure called a message. When information is to be transferred from one task to another, the sending task sends a message containing the information to an exchange. The other task goes to the exchange and waits to receive the message. Each exchange has associated with it a queue for messages sent to the exchange by any number of tasks and a queue for tasks waiting to receive a message at the exchange. Delivery of messages at an exchange is handled in a first-in-first-out (FIFO) fashion. A task can receive a message at an exchange in three different modes. These modes pertain to the time that the task will wait for the message. A waiting task can either wait indefinitely for a message, wait for a specified number of system time units, or not wait at all. In the latter case the request for a message is handled by checking to see if there is a message available at the specified exchange during the time the request is made. If the message is available it is returned to the requesting task; otherwise the request is ignored.

This message exchange concept allows for the first task interaction given above, that of communication. Information can be passed from one task to another in the form of a message. The second interaction, synchronization, is also handled through exchanges. To illustrate, suppose task A reaches a point where it is to signal another task B to perform a function. If task B is waiting indefinitely at an exchange, task A can initiate the operation of task B's function by sending a message to that exchange. RMX-80 translates the occurrence of hardware interrupt into a message sent to a special exchange called an interrupt exchange. This allows a task's activity to be sychronized with external hardware events in the same fashion as with internal software activity. Sychronization with system timing is provided by the timed wait feature of message exchanges. The third task interaction, resource management, is handled through the use of exchanges by using a message reserved for a given resource. This "key" message is initially sent to an exchange also reserved for that resource. Tasks competing for the resource simply wait at that exchange for the message. With this protocol only one task will possess the message at any given time and therefore possess the key to that resource. Other tasks are prevented from using the resources until the "key" message is returned by a task using the resource, effectively saying it has completed its use of the resource for that time. The three fundamental interactions between tasks, communication, synchronization and resource management, are all provided for by RMX-80 through its message exchange facilities.

RMX-80 Nucleus

Operations in an RMX-80 installation center around an operating system known as the Nucleus. This software "core" of activities manages the entire system. Multi-tasking is supported by maintaining a record of each task's execution state. The RMX-80 Nucleus controls the allocation of CPU resources to tasks and performs the required housekeeping for starting and stopping task execution. The Nucleus manages all activity at exchanges including maintaining queues and providing timing of specified wait periods. It also maintains the interrupt system by sending messages in response to hardware interrupts to an interrupt exchange. Additional general task management functions are supplied as part of the RMX-80 Nucleus. Functions are provided for the creation and deletion of tasks and exchanges in the executing system. The suspension and resumption of tasks are also accommodated. Initialization of the executing system is provided for so as to allow a predetermined set of tasks and exchanges to be defined and initialized for a system start-up. Tasks in RMX-80 have an assigned priority that is a measure of the importance of the task's operation. The Nucleus recognizes the priority of tasks and schedules their execution accordingly.

There are eight principal services provided by the RMX-80 Nucleus in the form of callable procedures. Their names and basic functions are the following:

1. RQSEND - Send a message to an exchange.
2. RQWAIT - Wait for a message at an exchange.
3. RQSUSP - Suspend a task.

4. RQRESM - Resume a suspended task.

5. RQCTSK - Create a task.

6. RQDTSK - Delete a task.

7. RQCXCH - Create an exchange.

8. RQDXCH - Delete an exchange.

Through the use of these system services and other task and message management facilities of the RMX-80 Nucleus, an entire system can be accommodated by RMX-80. All the requirements of a real-time, multi-tasking system as described in the previous section are met:

- o There can be multiple tasks.
- o Each task has an associated priority.
- o Tasks can be stopped by waiting at exchanges, use of the suspension service, or slow I/O delays.
- o When one task is stopped another can be started.
- o Tasks are allowed to interact through the use of exchanges and messages to accomplish a variety of ends.
- o The executing system can respond to an external event.

The RMX-80 Nucleus acts as a combination task traffic cop and arbiter, coordinating the concurrent activities of an executing RMX-80 based application system. The most important feature of a software installation that utilizes RMX-80 is that the system is event driven where the driving events are external, in the form of interrupts, and internal in the form of messages exchanged between tasks under a system timing scheme.

RMX-80, in addition to providing the Nucleus as the application's operating system, provides pre-developed optional tasks for performing

some common duties associated with a real-time, multi-tasking installation. These tasks include a Terminal Handler, a Free Space Manager, a Disk File System, a Bootstrap Loader, Analog Handlers, a Command Line Interpreter, and a Debugger. The resources or services provided by these tasks are accessed by application tasks through the use of the RMX-80 message exchange facilities. Only two of these optional tasks were utilized in the RDADS application, the Terminal Handler and the Command Line Interpreter. The function and operation of these two tasks will be discussed individually in the following sections. For more detailed information on RMX-80 consult the RMX-80 User's Guide [20].

RMX-80 Terminal Handler

A software interface between application tasks and an operator's terminal is provided for in an RMX-80 based installation by the RMX-80 Terminal Handler. The Terminal Handler is supplied as a collection of tasks that support basic and advanced serial communications through some type of terminal console. Although this support is directed primarily towards use as an interface between an operator and the executing system, it is not limited to this. A smart device such as another processing system can be substituted for the operator in the interface scheme. Therefore, the terminal Handler provides a driver for interfacing the RDADS system to either the Data Van Computer of the RPS or to a stand alone terminal console.

From the viewpoint of the programmer, use of the Terminal Handler is accomplished through special exchanges associated with terminal

input and output. When a task wants to receive data from the terminal input stream, it sends a request message to a special input-request exchange. The Terminal Handler responds by returning input data to the requesting task in the form a message sent to an exchange specified by the requesting task. In a similar fashion, a task wanting to output data to the terminal output stream would send a request message to a special output-request exchange. The Terminal Handler transmits the data and returns to the requesting task a message that signals the completion of the output operation.

The RMX-80 package supplies four different versions of the Terminal Handler. These provide for 1) full input and output operation, 2) full output only operation, 3) minimal input and output operation, or 4) minimal output only operation. The RDADS installation required both input and output operations. Since the minimal version used considerably less code space than the full version, which provided no advantages for this application, it was decided that the minimal version would be used. Therefore the minimal input and output operation version of the Terminal Handler was incorporated in the RDADS installation to act as the interface to the either target device, the remote Data Van computer or a local console terminal.

The input portion of the Terminal Handler provides for basic line input to the system with some useful line editing features. Additional detail on the Terminal Handler's input facilities are presented in the next section on the Command Line Interpreter. The output portion of the Terminal Handler provided for basic string output to the target. In order to provide some additional output utilities and formatting

functions, a package of software procedures was developed and labeled the Remote Operations Data Formatter And Transmission (RODFAT) package. More specifically, RODFAT provided routines for the formatting and display of character strings, numeric values, and other various types of displayed elements such as lines and boxes. It also provided specific routines for the display or output of command response data. In order to accomplish many required data conversions and manipulations, a second package of general system utilities was developed. This package was actually developed as a procedure library labeled the System Utilities (SYSUTL) Library. Conversion routines supplied by the library provided for converting data between various representations such as ASCII, hex, decimal, and ASCII encoded hex, decimal and binary. Another important procedure supplied in the SYSUTL Library was a routine that provided a simple interface to the RMX-80 Terminal Handler output facilities. This routine called DISPLAY made it possible for an application to output data through the Terminal Handler via a single procedure call. The source listings for both the RODFAT Utilities Package and the SYSUTL Library are included in Appendix B. These listings contain extremely adequate comments for explanation of the procedures and data structures included in them.

RMX-80 Command Line Interpreter

The RMX-80 Command Line Interpreter (CLI), supported through the RMX-80 Terminal Handler, was designed to allow an operator at the system console to initiate activities of user supplied application tasks. This is accomplished by entering a keyword command followed by an

optional command tail at the terminal. The Command Line Interpreter compares the command keyword with entries in a command table. When a match is found the table identifies an exchange assigned to that command. A message, containing the command tail if entered or a null tail indication, is sent to that command exchange to signal to the command task that the command was entered at the console. The command task executes a specific function either on the basis that the command alone was entered or optionally as directed by the command tail. For each and every command task in the system there is an associated command keyword and command exchange in the Command Line Interpreter's command table.

In the RDADS installation the command tasks are those already identified earlier in the discussion of the RDADS design. Each of the command task's command keywords were incorporated in the Command Line Interpreter's command table along with an assigned exchange. Each command task was developed as a program task that executes under RMX-80 Nucleus control. Every command task follows a general logical flow pattern that is common to all the command tasks. This logic is best illustrated as follows:

- o Initialize the task's software data structures and hardware constituents.

  .
  .

- o Begin a DO FOREVER loop:
  - o Wait for the input of the command via the Command Line Interpreter.

- o Interpret the command tail.

  .
  .

- o Perform operations based on the command tail interpretation and other pertinent factors.

  .
  .

- o Send data response to console terminal.

  .

  .

- o Send message to Command Line Interpreter that signals completion of the commands execution.

- o End of loop.

The first part of the task simply initializes the command's data variables and the hardware that the command controls. The second part of the command task is a loop that is executed once for each time the command keyword is entered at the terminal.

The Terminal Handler directly supports the Command Line Interpreter. In fact, when the CLI is included in an RMX-80 installation, all terminal input is directed to the CLI. It has already been noted in the previous section that the Terminal Handler acts as the interface from the RDADS to either the data van or a local terminal. It should be evident then that the source of command input to the Command Line Interpreter can be the remote computer in the RPS Data Van or the local terminal attached in its place.

Input from the terminal is supplied to the CLI in the form of a logical line. A logical line is a series of characters terminated by a break sequence which is usually a carriage return and line feed. This equates to a line entry from the input source such as the console keyboard. When the CLI is used the line entry is a command line. The CLI interpreter tries to identify the first word in the command line as a command keyword. This is done by finding a match with an entry in its command table from which it can identify an associated exchange. The CLI would then send a message to that exchange to inform a command task that a command keyword had been entered. The remainder of the command line (the command tail) is sent in this message to the receiving command task. It is the job of that task to interpret the meaning of the command tail. The command tail, as discussed in the section on Command Tasks previously, is a means by which options of a specific command are specified to the task. To facilitate the interpretation of the command tail, a set of functions and procedures were designed. They were developed and implemented as a package of utilities named the Remote Operations Command Tail Interpretation Package which is discussed in the next division of this section.

Analog Input Handler

In order to obtain analog data from the IF receiver, a software driver for the iSBX-311 Multimodule Analog Input board was required. Although the RMX-80 Executive package provided analog I/O handlers, they could not be used since they did not support the Multimodule board. An analog Input Handler task similar to those provided by

RMX-80 was written for driving the iSBX-311. This task was labeled AIHTSK (for Analog Input Handler TaSK). AIHTSK was designed to service requests for analog input data made through a reserved exchange, AIREQX (for Analog Input REQuest eXchange). Request messages sent to this exchange followed the standard format of an RMX-80 message as outlined in the RMX-80 User's Guide. Documentation of the logical operation of this task and guidelines for its use are included as comments in the source listing. This source is included in the software listings of Appendix B.

Utility Packages and Libraries

Several routines were developed as callable procedures or functions to provide for some needed software operations. Hardware communications in the RDADS were handled by special routines developed for that purpose. These routines were included in a software library labeled the RDADS Utilities Library. Utilities for general data conversion and manipulation were developed and stored in a library labeled the Systems Utilities Library. Interpretation of command tails was provided by a package of utilities called the Remote Operations Command Tail Interpreter. Finally, formatting of data to be displayed on a console terminal or to be sent to the data van CPU was accomplished using a package called the Remote Operations Data Formattor. Each of these four packages will be discussed in the following sections where all the routines they provide will be itemized.

RDADS Utilities Library

This library provides routines for communications with the RDADS hardware, including the RF transceivers and the IF Controller. It also included general analog communications which was accomplished by including the AIHTSK routine described previously.

RF head communication routines were developed as a single source file labeled HEDCON.SRC (for head control). All RF head control and monitoring is accomplished through routines provided by this file. Routines developed included:

1. INIHED - Sets up the programmable peripheral interface (PPI) used for communicating with RF Heads and initializes the ports it controls.
2. RXDCON - Sets the receiver polarization transfer switch to the specified polarization.
3. TXPCON - Sets the transmitter polarization transfer switch to the specified polarization.
4. RFOCON - Sets Reserved RF Control 0 signal as specified (On or Off).
5. RF1CON - Sets Reserved RF Control 1 signal as specified (On or Off).
6. GENPAT - Generates a bit pattern used in setting a control port or enabling an indicator circuit.
7. ENBIND - Enables a specified indicator circuit for a specified radar head.

8. GETPOL - Reads a specified polarization indicator circuit for a specified radar head to determine the state of that polarization transfer switch.
9. GETCAL - Reads the calibration indicator circuit for a specified radar head to determine the state of that calibration transfer switch.
10. COPCON - Set the Cal./Op. Transfer switch of a specified radar head.

More detailed explanations of each of the routines and guidelines on their use can be found in the source listings in Appendix B.

The IF I/O Driver routines source file, IFIODR.SRC, provided all necessary communication drivers for interfacing to the IF Controller board. Routines found in this source file are:

1. INIFIO - Sets up the PPI used for communicating with the IFC board, initializes these parallel ports, and issues a reset signal to the IFC.
2. IFGSEL - Sets the IF Group Select lines to select a specified parameteric register-buffer pair.
3. IFINEN - Places IFC into transmit mode to enable the currently selected transmitter (parametric buffer).
4. IFSEON - Turns on both the primary and secondary Strobe/Enable signals to the IFC.
5. IFSEOF - Turns off both the primary and secondary Strobe/Enable signals to the IFC (complement function to IFSEON).

6. IFDBTX - Sets the IFC Data Bus (port) into the transmit mode of operation (configures the port for data output to IFC).
7. IFDBRX - Sets the IFC Data Bus (port) into the receive mode of operation (configures the port for intput of data from the IFC).
8. IFWRIT - Outputs a data byte to a specified target parameteric register on the IFC.
9. IFREAD - Inputs a data byte from a specified source parametric buffer on the IFC.

Of these routines IFWRIT and IFREAD are the most important since they are the only routines called for the output or input of IFC data. All of the other routines with the exception of INIFIO are used by IFWRIT and IFREAD to accomplish their operations. Source listings of all the above routines can be found in Appendix B.

All of the routines described in this section were included in the final Remote Operations Installation by linking the RDADS library with the main program module. The library was labeled RDADS.LIB and was created under the Intel ISIS-II Librarian facilities.

System Utilities Library

Utilities for the conversion and manipulation of data in the ROU were included in a library file called SYSUTL.LIB. Routines in this library provide functions for converting between different data types such as ASCII, hexadecimal, and decimal and for manipulating data

between different storage variables such as nibble, byte, address, and string variables. Also the package provides a simple to use interface to the RMX-80 Terminal Handler's output facilities. This library's routines were useful mainly in the formatting of input and output data. They are:

1. ASCLO - Converts the digit in the low nibble of a byte value to its ASCII code.
2. ASCHI - Converts the digit in the high nibble of a byte value to its ASCII code.
3. ASCII2 - Converts two digits expressed in a byte value to a pair of ASCII codes.
4. ASCII4 - Converts four digits expressed in an address value to a string of four ASCII codes.
5. PACK - Combines two nibbles (digits) into a single byte value.
6. COMB - Combines two byte values into a single address value.
7. HEX - Converts an ASCII code into a single Hexidecimal digit (0-F) with a check for valid inputs.
8. B2D - Converts a Binary integer into a Binary Coded Decimal (BCD) integer. Input and output are both address values.
9. D2B - Converts a BCD integer into a Binary integer. Input and output are again both address values.
10. BLANK - Scans an ASCII buffer for leading zeros and converts them to blanks.

11. DISPLAY - Outputs an ASCII buffer of specified length to the RMX-80 Terminal Handlers output stream.
12. INITDSP - Initializes the data structures used by DISPLAY.
13. CROUT - Outputs a Carriage Return and Line Feed via DISPLAY.

Source listings for each of these routines can be found in Appendix B. The listings document the use and operation of the routines in detail.

Remote Operations Command Tail Interpreter

The RMX-80 Command Line Interpreter actually interprets only the command keyword of the command line. The remainder of the command line, the command tail, is passed to the command task for it to process. To facilitate the processing of command tails the Remote Operations Command Tail Interpreter (ROCTIP) was developed. This is a package of procedures that processed an ASCII buffer to do such things as search for the occurrence of keywords or convert ASCII information to flag settings or numeric data. It was developed as a single source file labeled ROCTIP.SRC.

ROCTIP processes a command tail by first setting its data pointers and length variables to correspond to the current command tail. The package provides a routine to accomplish this. Other routines are then used to scan the command tail buffer for pertinent information. The routines included in ROCTIP are the following:

1. RST$CCLIB - Resets pointers and lengths as specified by

information obtained from the RMX-80 Command Line Interpreter.

2. SAVE$SAB - Saves the current buffer position and character count.
3. RESTOR$SAB - Restores previous buffer position and character count saved by SAVE$SAB.
4. SCAN - Advances to next character position.
5. SCAN$BAK - Backs up to previous character position.
6. SKIP - Advances past blanks to first non-blank character.
7. CCLIB$MT - Checks to see if buffer is empty.
8. VALID$DEL - Checks for occurrence of a valid delimeter (blank or carriage return).
9. SEARCH - Scans the buffer for the occurrence of a specified string or a valid abbreviation of the string.
10. SPOTEQ - Checks for the occurrence of an equal sign in the buffer.
11. SPOTQM - Checks for the occurrence of a question mark in the buffer or equivocally an empty buffer.
12. SCAN$NUM - Scans the buffer for an ASCII string representing a numeric data value and returns the value found.
13. ID$HED - Checks for the occurrence of a head specifier (RF Head number or Band character).
14. REC$CAL - Checks for the keyword "CALIBRATE" or any abbreviation of it.

15. REC$OP - Checks for the keyword "OPERATE" or any abbreviation.
16. REC$CONT - Checks for the keyword "CONTINUOUS" or any abbreviation.
17. REC$SING - Checks for the keyword "SINGLE" or any abbreviation.
18. ID$POL - Checks for and identifies an optional polarization specifier in the command tail.

All of these routines were utilized in processing the command tails of the command task itemized earlier. Their design directly supports conditional execution on the results of their operation. This made logical design of the command tasks processing simple and straightforward. Complete details on the ROCTIP routines can be found in the listing in Appendix B.

Remote Operations Data Formatting and Transmission

Output of information from the RDADS as stated earlier was intended to be received by either the Data Van CPU or a console terminal. These two setups present very different operational requirements. Output for the first setup is processed at the receiving end by the data van CPU and should be concise. This output requires very little special processing and needs to be human readable only enough to allow for ease in testing and trouble shooting. This means that data can be transmitted in any form as long as it is ASCII encoded to allow human recognition at a terminal. Data output for the second setup is very different. This output is intended to ease testing, trouble

shooting, maintenance, and calibration. Therefore, it should be easily viewed and interpreted by an operator.

The Remote Operations Data Formatting and Transmission utility (RODFAT) provided routines that output command response information for either communication setup. Each routine's actions are dependent on the current setting of the communications mode set during operation. Output provided by them follows the format specified earlier in the description of the command tasks. Other routines are included in the formattor for the display of boxes and labels on the console terminal when in the LOCAL mode of operation. Also included are routines that display the current status information in reserved locations on the console which supports the command response output in LOCAL mode. These routines were developed as a single source file labeled RODFAT.SRC. Routines included in RODFAT are the following:

1. BOX - Draws a box of specified length and height at a specified console screen location.
2. DSP$NUM - Displays the decimal representation of a specified hex value at a specified console screen location.
3. DSP$RANGE - Outputs the current Range Delay values for both Calibrate and Operate modes (format dependent on current communication mode).
4. DSP$GATE - Same as DSP$RANGE except for Gate Delay values.
5. DSP$MODE - Outputs the current radar mode setting (comm. mode dep.).

6. DSP$GAIN - Outputs the current gain value (comm. mode dep.).
7. DSP$HEAD - Outputs the current RF Head selected (comm. mode dep.).
8. DSP$POL - Outputs current polarization for specified RF circuit section (comm. mode dep.).
9. CMD$ERR - Outputs "Command Tail Error" message.
10. INI$CRT - Initializes console terminal by displaying logo, boxes, labels, and current parameter settings.

There are other routines included in RODFAT but they are intended only for use internally to that package. They are not listed here but do appear in the source listing of RODFAT in Appendix B. All RODFAT routines can be found in this listing along with comments on their use and function.

Remote Operations Main Utility

All of the command tasks identified earlier are defined in the Remote Operations Main Utility (ROMAIN) along with the Radar Task. The source file is under the name ROMAIN.SRC and a listing appears in Appendix B. All of the utility packages and libraries discussed above were linked with this main utility along with the necessary RMX-80 Libraries to provide the final software installation.

All software developed for the RDADS installation was designed using structured programming techniques. The approach was to develop an operating system, namely the Remote Operations Utility, that consisted of command and operation tasks. These tasks accomplished all

the necessary functions of the RDADS. Development of the ROU was facilitated by the use of RMX-80, a Real-Time Multitasking Executive package. Individual subprogram and function utility packages were developed for hardware interfacing, data conversion and manipulation, command line interpretation, and data formatting and display.

PROJECT SUMMARY

The design and implementation of the Remote Data Acquisition and Distribution System (RDADS) have been described. The RDADS is a subsystem of the RPS experimental remote sensor developed by the Remote Sensing Center at TAMU. This sensor is a two truck system that is used in field experimentation for soil moisture remote sensing.

The RDADS is a stand alone processor system that supports operation under the direction of a main computer. The system also allows connection of an ANSI standard ASCII CRT terminal (CIT-101 or equivalent) in place of the main computer for testing, calibration, and maintenance. In either configuration, a set of predefined operation commands allow for the interactive control of the radar's electronics. The RDADS performs complex operations for the user through the use of predeveloped hardware and software functions.

The RDADS hardware features an 8085 microprocessor based single board computer, special analog I/O interface boards, and a dynamic RAM board installed in an industrial standard computer card cage. This hardware makes up a stand alone computer labeled the RDADS processor. The hardware used in the RDADS processor provides flexibility and expansion capability to allow changes in the system architecture. A special high-speed digital electronics board was designed and developed to control range-gating for the radar. This IF Controller board, uses Schottky TTL integrated circuits to provide a time resolution of 16.67 nS for the range-gating function. The IFC also provides other program-

mable functions for controlling the transmission and reception of radar signals through the IF section of the radar. The RDADS processor controls the IFC and other radar hardware through parallel digital I/O ports.

Software developed for the RDADS processor defines most of the functions that the RDADS performs. This RDADS software was developed as modular blocks. Each module provides either independent tasks or shared support utilities. New tasks and support utilities are easily added without changing existing modules. Individual existing modules are also easily modified without changing or affecting other existing modules. RMX-80, a real-time, multi-tasking, executive package, is used as the operating system software to control the system tasks.

Future Considerations and Recommendations

The IF receiver gain is controlled by the RDADS through a software algorithm. This algorithm is a Digital Automatic Gain Control (DAGC) function. Due to the way the DAGC algorithm is implemented, a total of 234 analog measurements are required for each radar data point obtained. This limits the radar data acquisition rate to one data point every 1.4 seconds. If this proves to be too slow for future needs, several steps can be taken to improve the DAGC's operation. Analog input signals are averaged by software at the rate of eight analog conversions per return value. This averaging is a software filtering of the analog input signal. A hardware filtering approach could possibly be implemented to speed up the radar acquisition rate by as much as 8 times faster. Another approach would be to modify the

DAGC algorithm to allow it to track the receiver gain. If each execution of the DAGC algorithm utilized the gain value obtained by the previous cycle, the number of analog acquisitions per DAGC iteration could be reduced. The overall radar acquisition rate could be improved by a factor of up to 26. The drawback to this approach is that the system would not be able to track large changes in received signals. The current approach allows any changes that stay within the dynamic range of the radar receiver. Limiting the response of the system to small perturbations in signal level may be desirable, however. Such an approach would provide an inherent filtering in the receiver and possibly improve the system's noise rejection. The improvement in acquisition rate that can be obtained by a tracking DAGC is limited by the amount of change in radar signal level that the algorithm is designed to accommodate. If the allowable change is limited to 1/2 dB the rate could be improved by the upper limit of 26 times faster.

Almost every hardware component that is interfaced to the RDADS processor has some type of feedback or is capable of being monitored. Software was written to support these hardware monitor functions at the interface driver level. The upper level commands, however, do not support these monitor functions. Included are the monitoring of supply voltage levels, checking of RF switch settings, and confirmation of data written to the IFC. A future improvement to the system would be to enhance existing control software to automatically monitor hardware through these available feedback facilities. Making such changes would increase the reliability of the system's performance. Adding commands to the system to support testing and maintenance may also be a desired

future improvement. As an example, adding commands for writing directly to and reading directly from the IFC's I/O channels might prove useful in maintenance of the IFC and other IF section hardware.

In general the system performs well in accomplishing the tasks for which it was programmed. The flexibility of both the hardware and software allows easy modification of the existing system to enhance its performance or upgrade its capabilities. The RDADS was developed to provided the control and data acquisition functions needed to operate the Radar Polarimeter System. The system supports the operation of four radar bands, three of which are currently implemented. Also, since the RDADS is flexible and expandable, it could easily be used for operating new sensors and support equipment added to the boom truck in future expansions. Interfacing additional equipment could be accomplished at moderately low cost in hardware by using the currently available expansion slot in the RDADS processor's card cage. A combination I/O and EPROM expansion card would provide additional control and data acquisition channels and added program memory space. Software is easily modified to support new equipment by adding new modular program tasks that are controlled by the existing operating system software. If speed of processing is a limiting factor in a desired expansion or modification, a second single board computer such as the iSBC-80/24 could be added in the availabe card cage slot instead. The iSBC-80/24 has full Multibus/Multimaster bus arbitration logic on-board so that two processors could operate independent of each other. Shared access to the existing Dynamic RAM board could be utilized to pass information between the two processes, if this approach was chosen. The most

important point is that the system has more capability than is currently utilized and can be expanded to provide even more. This fact should not be overlooked in future planning and design of new sensors and support equipment.

REFERENCES

[1] Texas A&M Research Foundation Project No. 4242-4 "Development of Remote Sensing Techniques for Measuring Soil Moisture," sponsored by NASA - Goddard Space flight Center, NASA Grant No. NAG5-31.

[2] A. J. Blanchard and B. R. Jean (1981), "Antenna Effects in Depolarization Measurements," Technical Report RSC-119, Remote Sensing Center, Texas A&M University, College Station, Texas.

[3] A. K. Fung, (1966), "On depolarization of electromagnetic waves backscattered from a rough surface," Planetary Space Science, 14, 563-568.

[4] J. C. Leader, (1971), "Bidirectional scattering of electromagnetic waves from roughsurfaces," J. Appl. Phys., 42, 4808-4816.

[5] J. W. Rouse, Jr., (1972), "The effect of the subsurface on the depolarization of rough-surface backscatter," Radio Science, 7(10), 899-895.

[6] P. Beckmann, (1968), The Depolarization of Electromagnetic Waves, pp. 144-162, Golem Press, Boulder Colorado.

[7] A. J. Blanchard, (1977), "Volumetric effects in the depolarization of electromagnetic waves scattered from rough surfaces," Technical Report RSC-83, Remote Sensing Center, Texas A&M University, College Station, Texas.

[8] A. J. Blanchard, R. W. Newton, L. Tsang and B. R. Jean (1981), "Volumetric effects in cross polarized airborne radar data," Technical Report RSC-120, Remote Sensing Center, Texas A&M University, College Station, Texas.

[9] H. Hirosawa, S. Komiyama and Y. Matsozaka (1978), "Cross polarized radar backscatter from moist soil," Remote Sensing of Environment, 7, 211-217.

[10] F. T. Ulaby, P. P. Batlivala and M. C. Dobson (1978), "Microwave backscatter dependence on surface roughness, soil moisture, and soil texture, part I: bare soil," IEEE Trans. Geosci. Electron., Vol. GE-16, Oct.

[11] iSBC-80/24 Single Board Computer Hardware Reference Manual, (1981), Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

[12] iSBX-311 Analog Input Multimodule Hardware Reference Manual, (1981) Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

[13] iSBX-311 Analog Output Multimodule Hardware Reference Manual, (1981) Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

[14] Edward Yourdon, and L. L. Constantine (1979), Structured Design, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

[15] R. C. Camp, T. A. Smay, and C. J. Triska (1978), Microcomputer Systems Principles Featuring the 6502/KIM, Matrix Publishers, Inc., Portland, Oregon.

[16] L. P. Meissner and E. I. Organick (1980), Fortran 77: Featuring Structured Programming, Addison-Wesley Publishing Company, Reading, Massachusetts.

[17] J. K. Hughes, (1979), PL/1 Structured Programming, John Wiley & Sons, New York, N.Y.

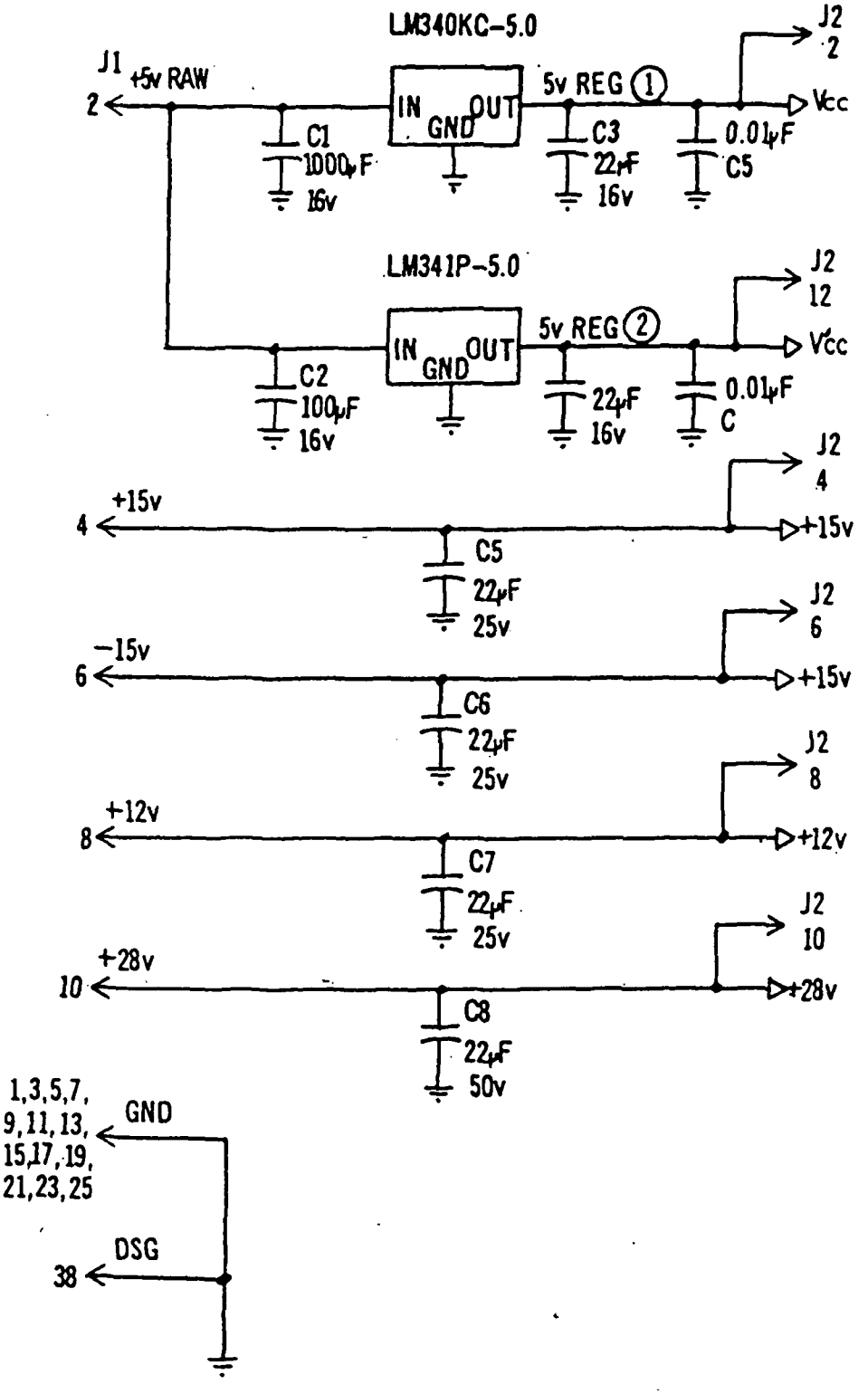
[18] R. J. Rader, (1978), Advanced Softward Design Techniques, Petrocelli Books, Inc., Princeton, N. Y.

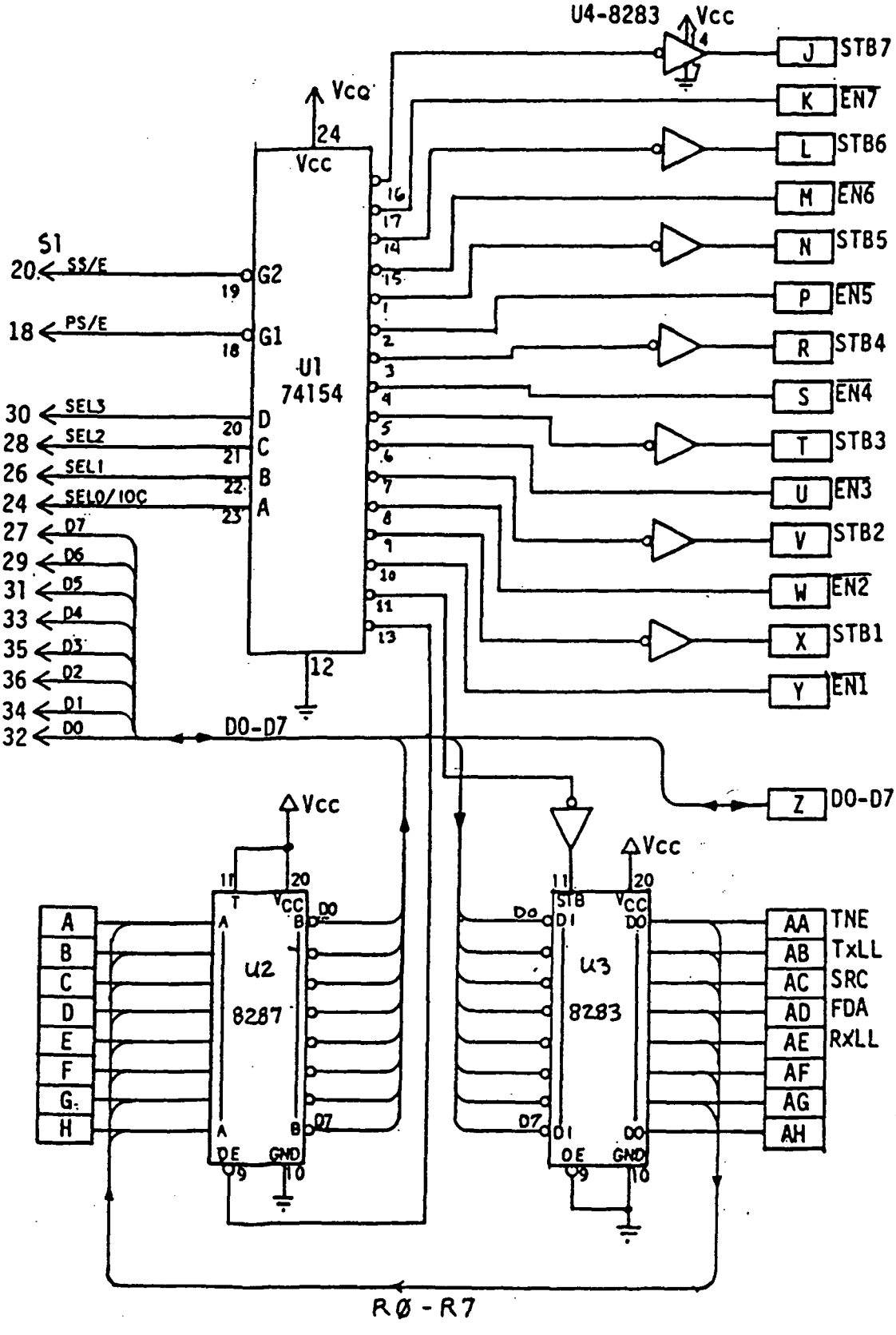
[19] Introduction to the iRMX 80/88 Real-Time Multitasking Executives, (1981), Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

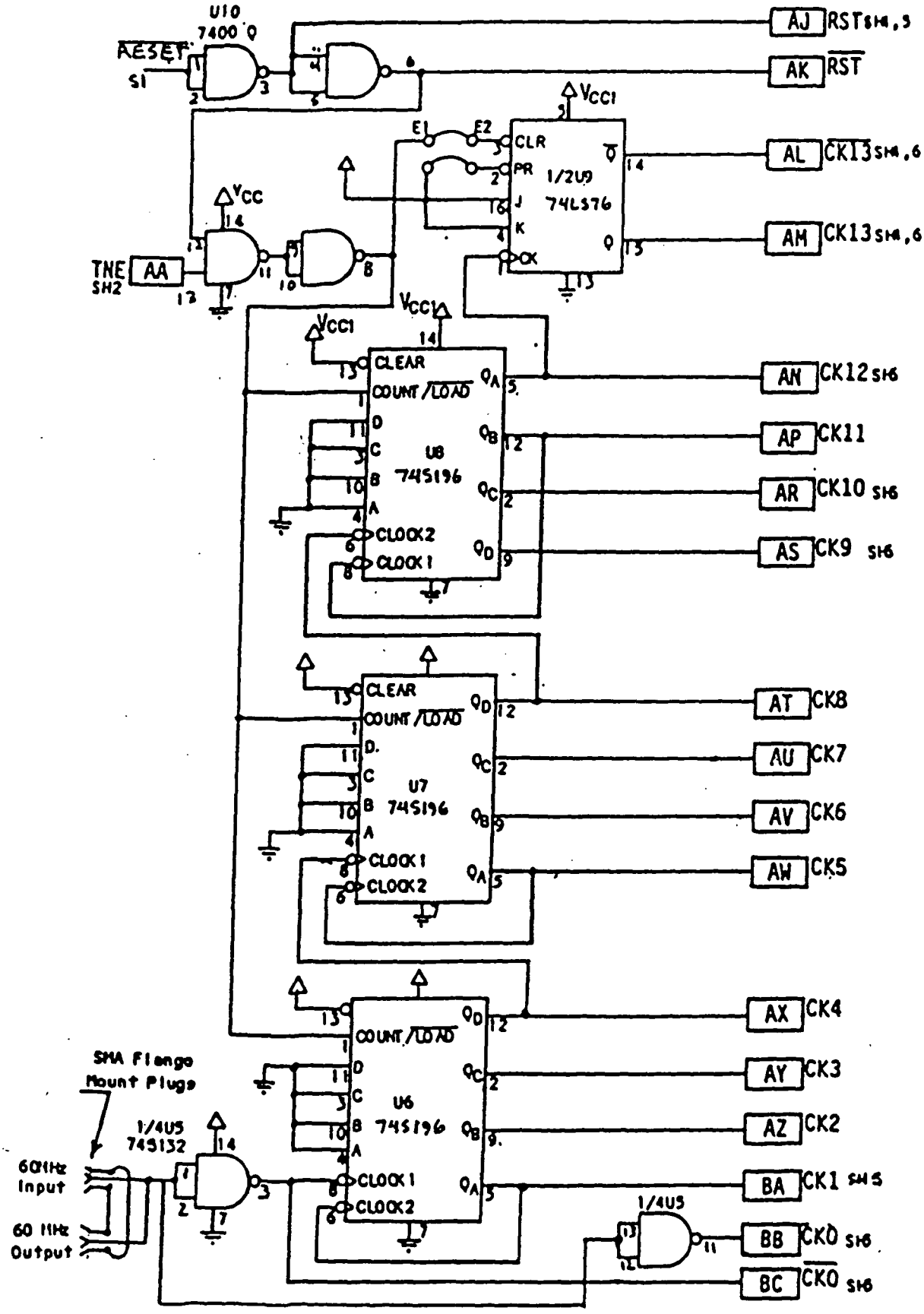
[20] iRMX-80 User's Guide, (1981), Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

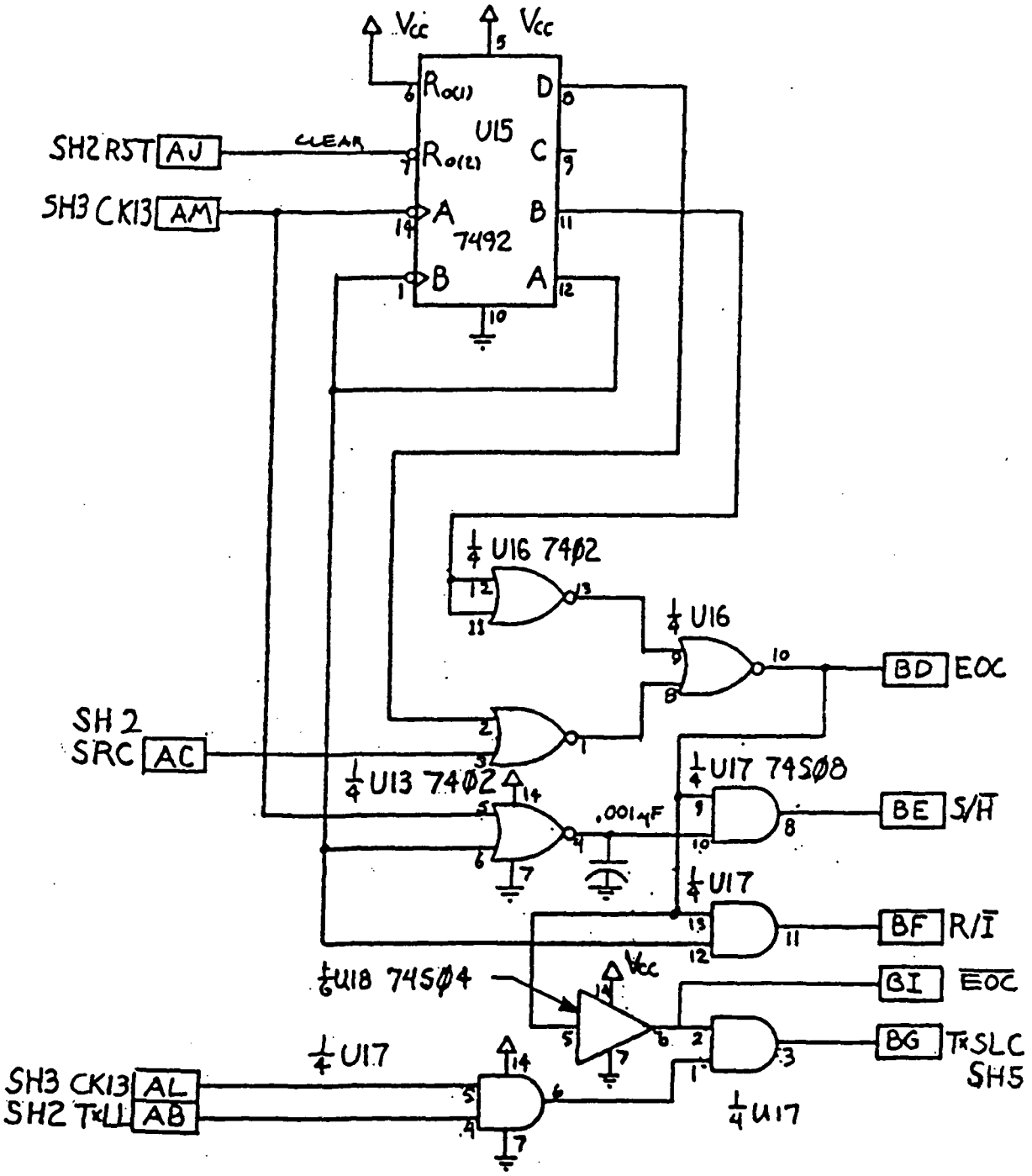
APPENDIX A

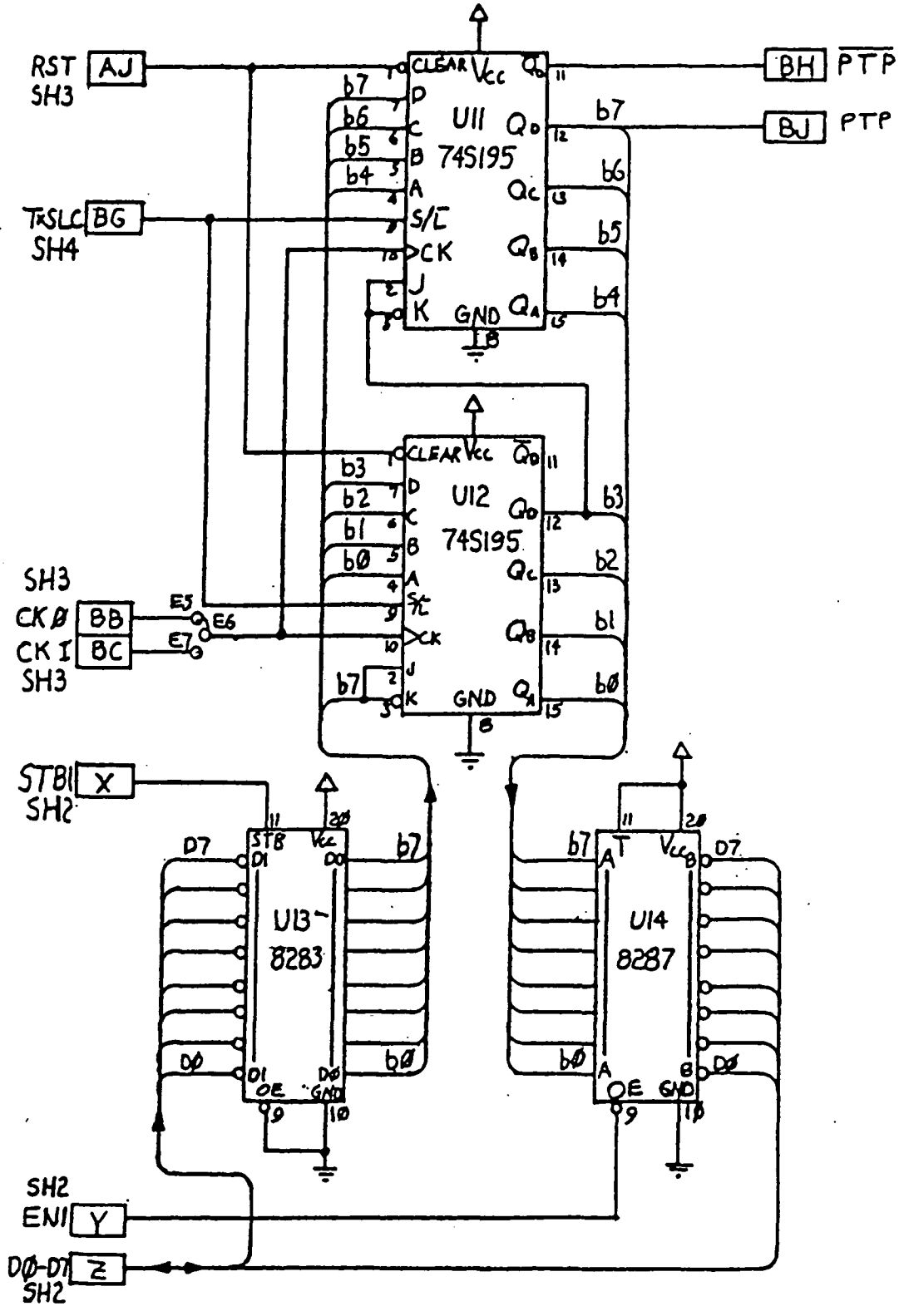
IF CONTROLLER CIRCUIT BOARD ELECTRICAL DIAGRAMS AND PRINTED CIRCUIT LAYOUT

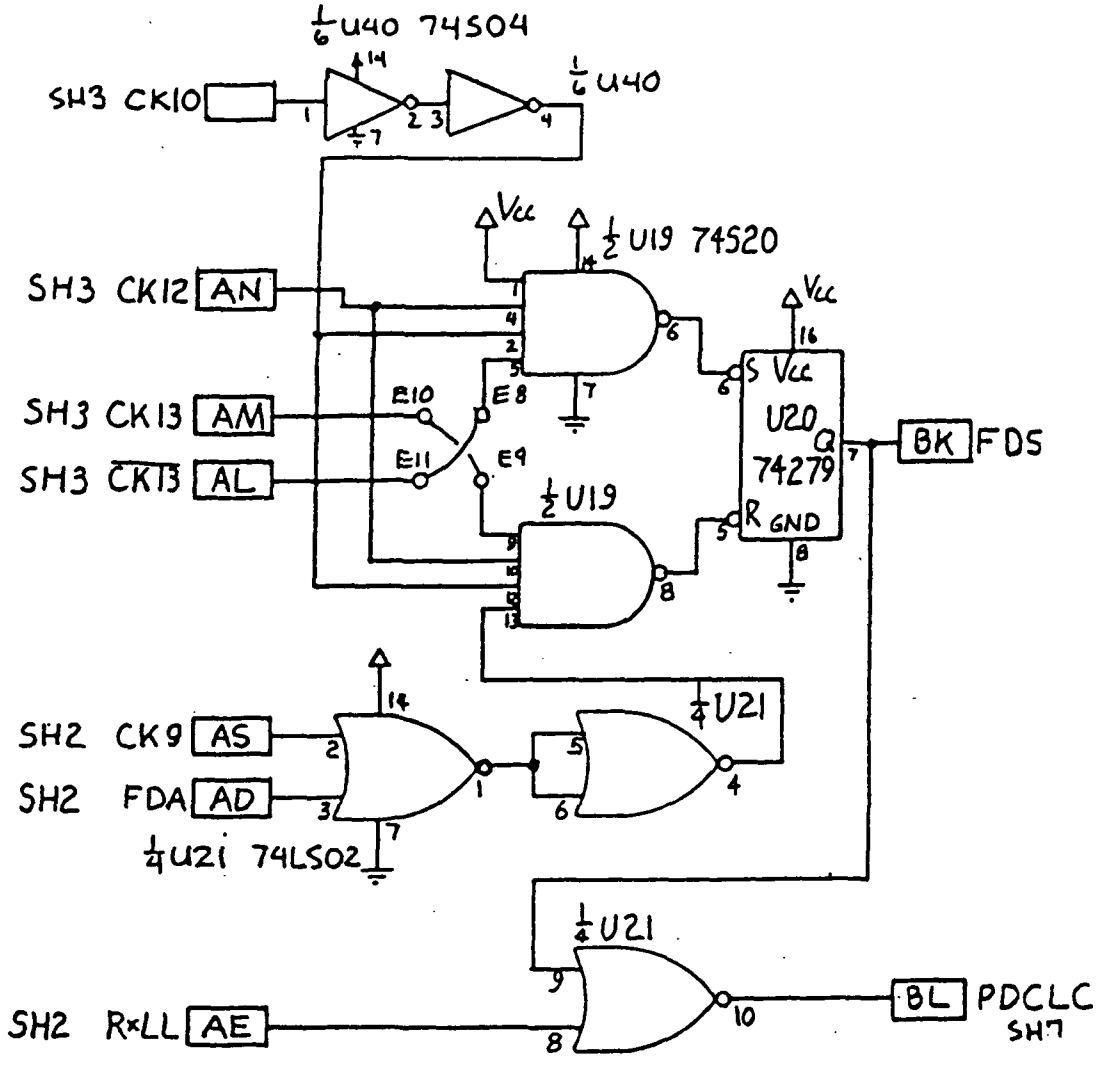

LM340KC-5.0
J1
+5v RAW
2
IN
GND
OUT
5v REG (1)
J2
2
Vcc
C1
1000μF
16v
C3
22μF
16v
0.01μF
C5
LM341P-5.0
J2
12
IN
GND
OUT
5v REG (2)
Vcc
C2
100μF
16v
22μF
16v
0.01μF
C
J2
4
+15v
4
+15v
C5
22μF
25v
J2
6
−15v
6
+15v
C6
22μF
25v
J2
8
+12v
8
+12v
C7
22μF
25v
J2
10
+28v
10
+28v
C8
22μF
50v
1,3,5,7,
9,11,13,
15,17,19,
21,23,25
GND
DSG
38

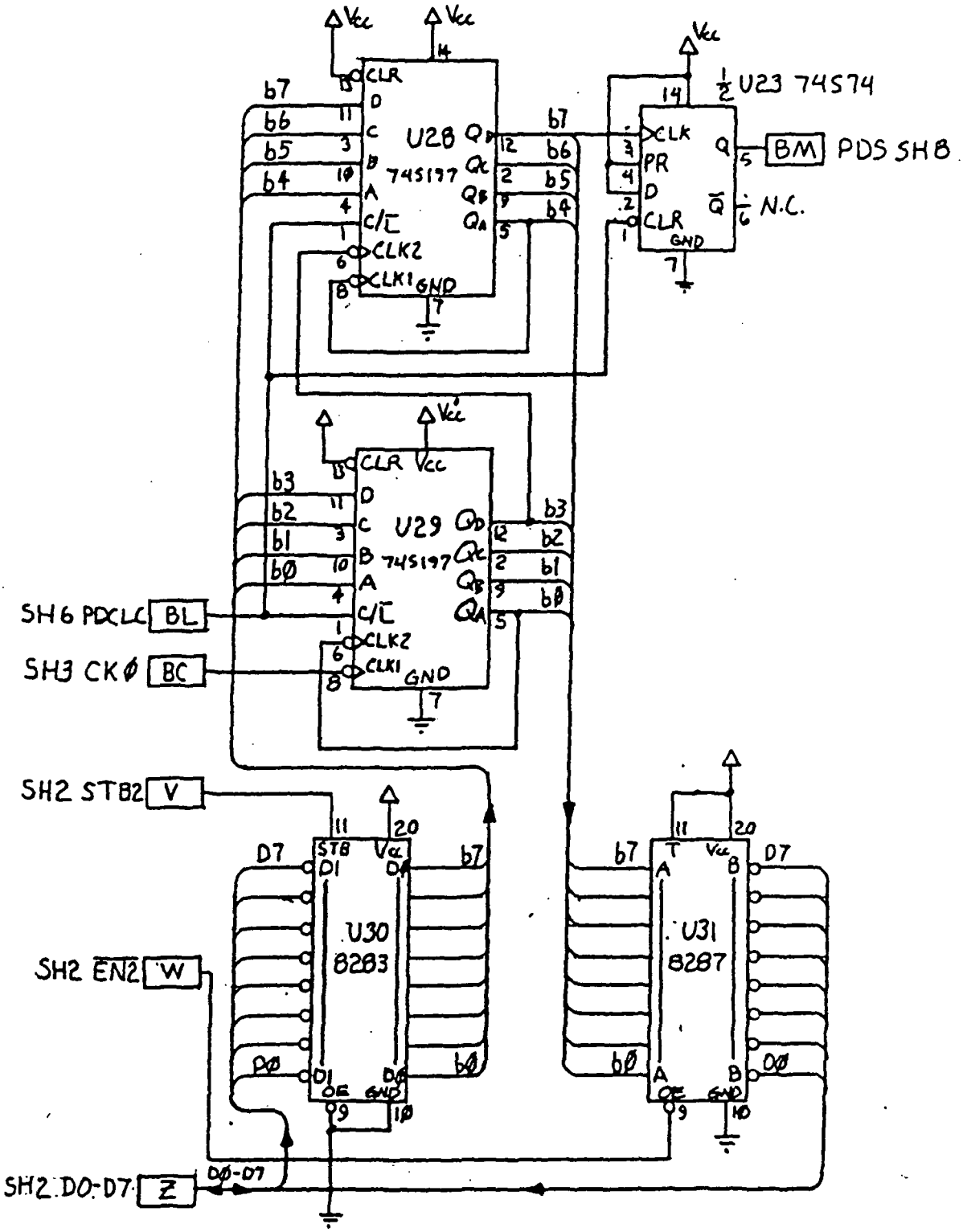

U4-8283
Vcc
J STB7
K EN7
L STB6
M EN6
N STB5
P EN5
R STB4
S EN4
T STB3
U EN3
V STB2
W EN2
X STB1
Y EN1
Z D0-D7
S1
20 SS/E
18 PS/E
30 SEL3
28 SEL2
26 SEL1
24 SELO/IOC
27 D7
29 D6
31 D5
33 D4
35 D3
36 D2
34 D1
32 D0
U1
74154
G2
G1
D
C
B
A
D0-D7
U2
8287
U3
8283
A
B
C
D
E
F
G
H
AA TNE
AB TxLL
AC SRC
AD FDA
AE RxLL
AF
AG
AH
R0 - R7

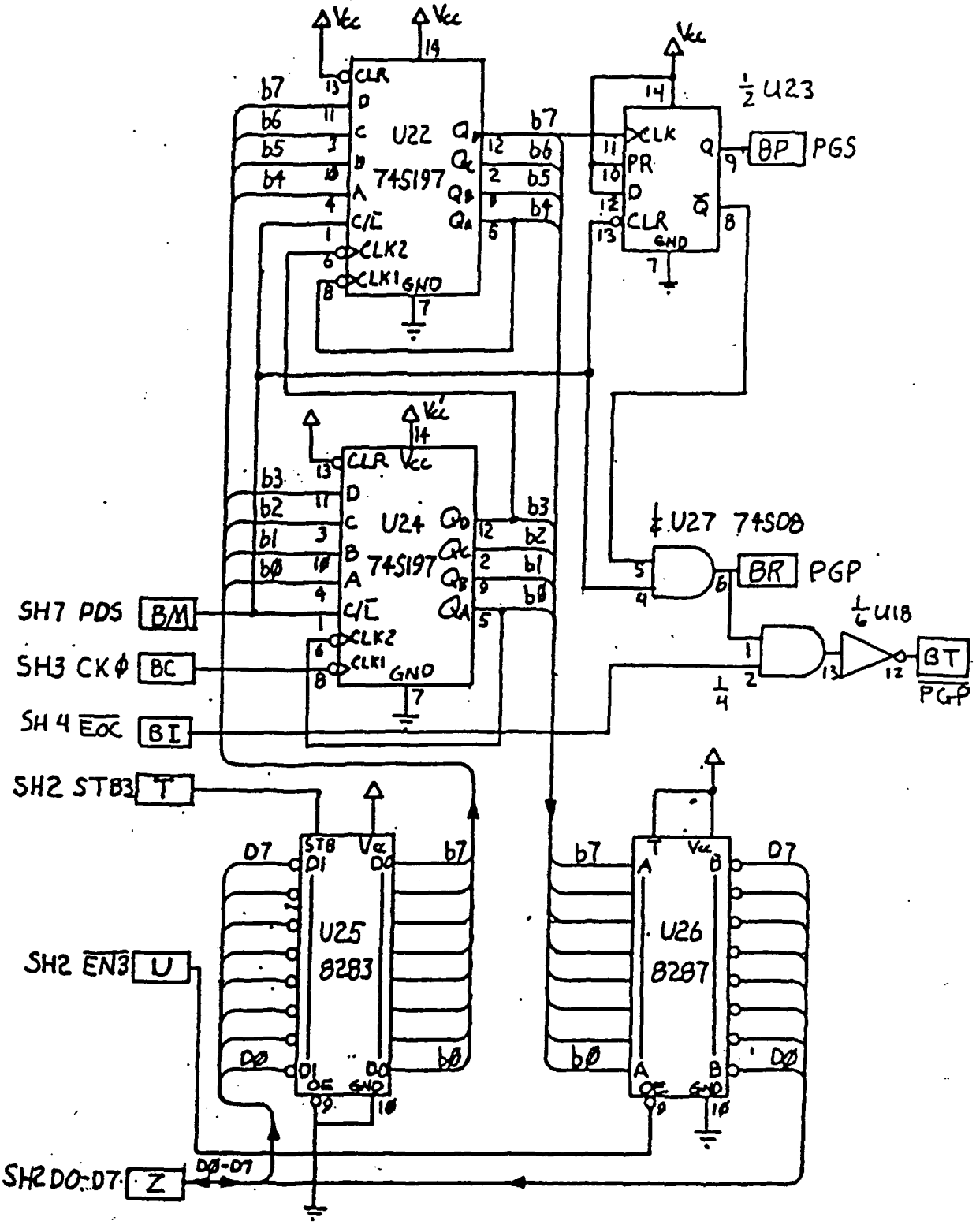

U10
7400
RESET
S1
AJ RST SH4,5
AK RST
VCC1
E1
E2
CLR
PR
J
K
CK
1/2U9
74LS76
AL CK13 SH4,6
AM CK13 SH4,6
VCC
TNE SH2
AA
VCC1
CLEAR
COUNT/LOAD
U8
74S196
CLOCK2
CLOCK1
AN CK12 SH6
AP CK11
AR CK10 SH6
AS CK9 SH6
U7
74S196
AT CK8
AU CK7
AV CK6
AW CK5
U6
74S196
AX CK4
AY CK3
AZ CK2
BA CK1 SH5
SMA Flange Mount Plugs
1/4U5
74S132
60 MHz Input
60 MHz Output
1/4U5
BB CKO SH6
BC CKO SH6

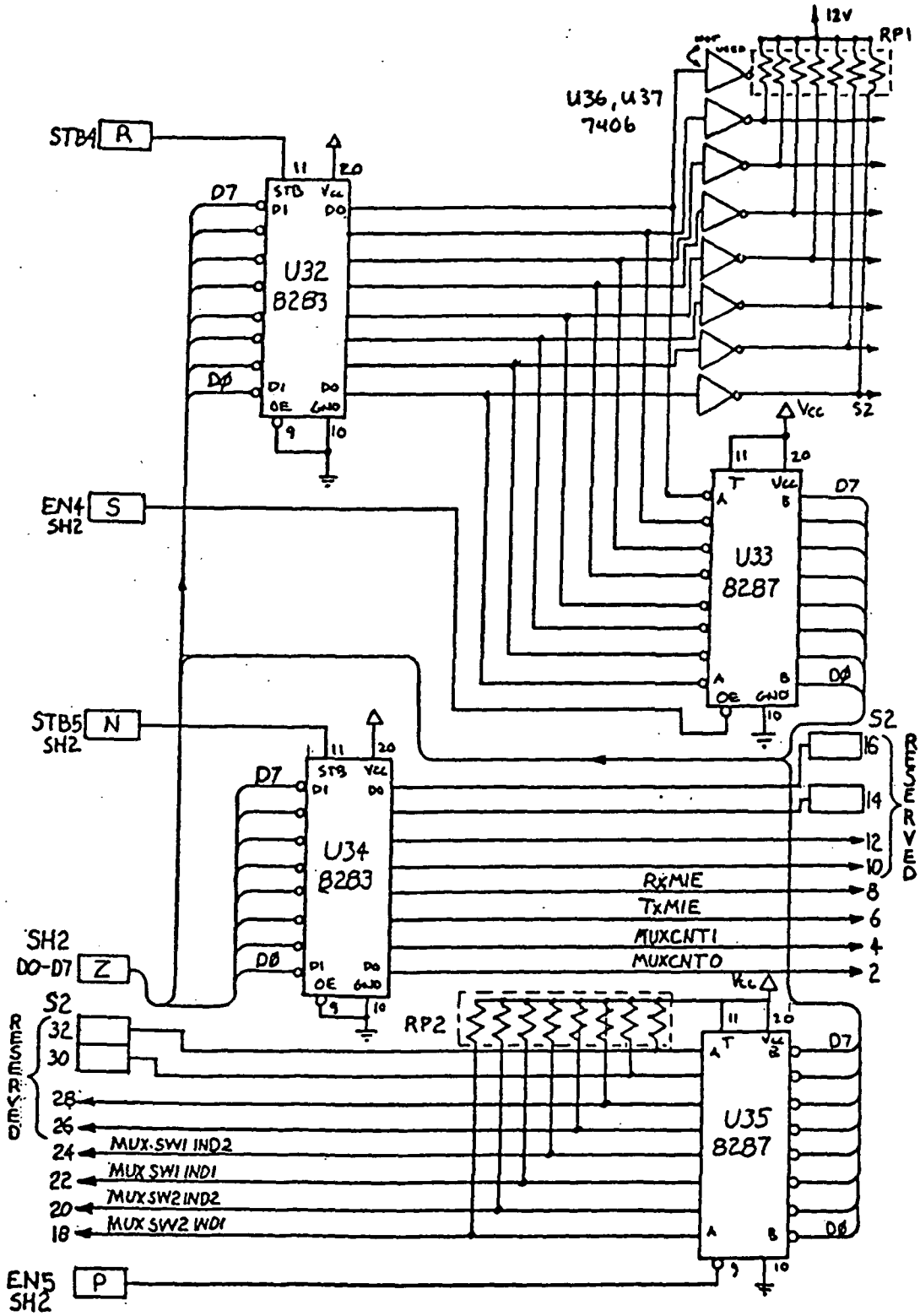

Vcc
Vcc
U15
7492
SH2 RST AJ
CLEAR
SH3 CK13 AM
1/4 U16 74Ø2
1/4 U16
BD EOC
SH 2
SRC AC
1/4 U13 74Ø2
.001μF
1/4 U17 74S08
BE S/H
1/4 U17
BF R/I
1/6 U18 74S04
BI EOC
BG TxSLC
SH5
1/4 U17
SH3 CK13 AL
SH2 TxLI AB
1/4 U17

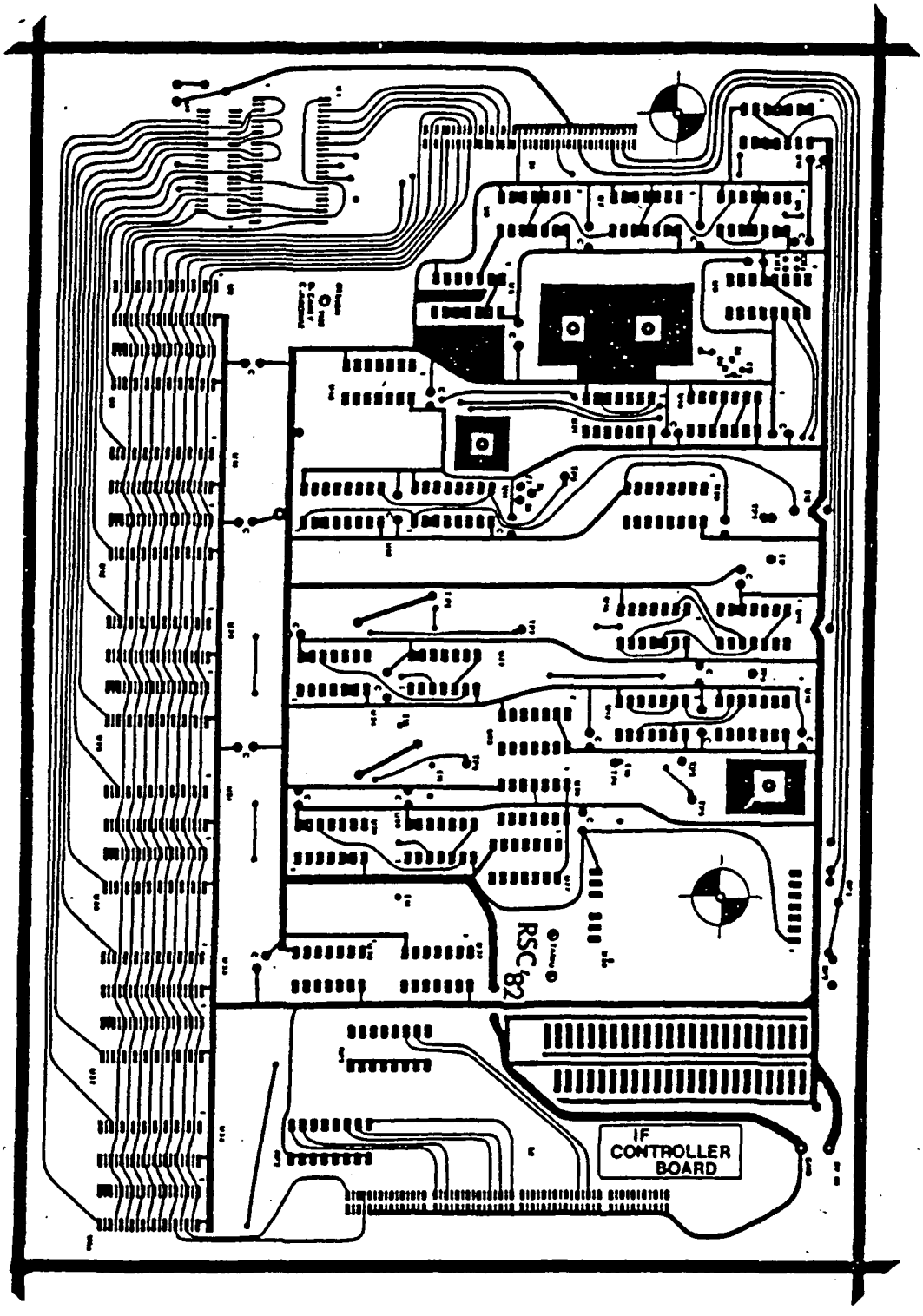

RST
SH3
AJ
TRSLC
SH4
BG
CK Ø
CK I
SH3
BB
BC
E5
E6
E7
STBI
SH2
X
SH2
ENI
Y
DØ-D7
SH2
Z
U11
74S195
U12
74S195
U13
8283
U14
8287
CLEAR
Vcc
GND
CK
S/L
J
K
STB
OE
BH PTP
BJ PTP
b7
b6
b5
b4
b3
b2
b1
bØ
D7
DØ

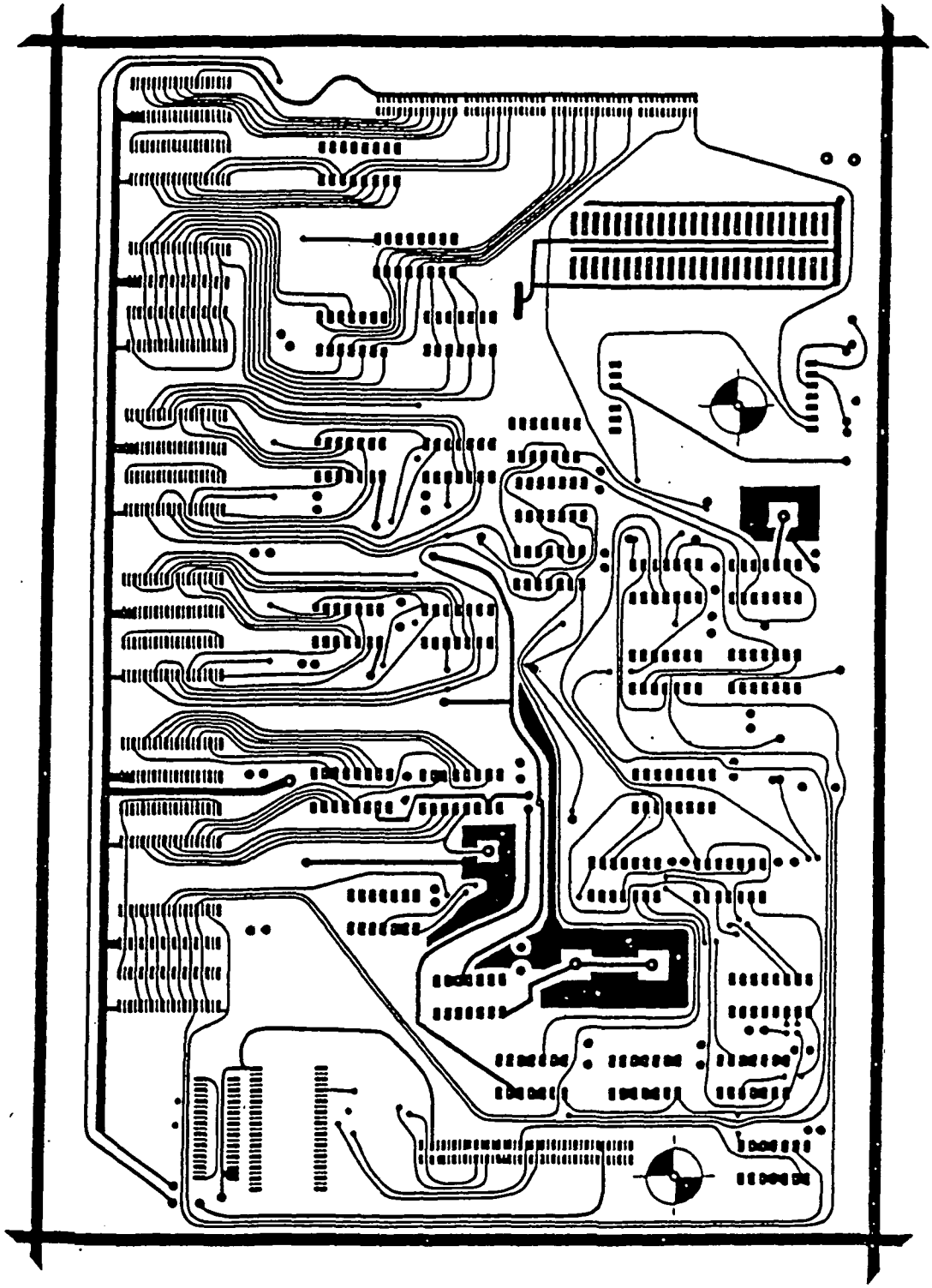

1/6 U40 74S04
SH3 CK10
1/6 U40
Vcc
1/2 U19 74S20
SH3 CK12 AN
SH3 CK13 AM
SH3 CK13 AL
E10
E11
E8
E9
1/2 U19
U20
74279
S Vcc
R GND
BK FDS
SH2 CK9 AS
SH2 FDA AD
1/4 U21
1/4 U21 74LS02
1/4 U21
SH2 R×LL AE
BL PDCLC
SH7


Vcc
U28 74S197
b7
b6
b5
b4
CLR
CLK2
CLK1
GND
1/2 U23 74S74
CLK
PR
D
CLR
Q
N.C.
BM PDS SH8
U29 74S197
b3
b2
b1
b0
SH6 PDCLC BL
SH3 CKØ BC
SH2 STB2 V
SH2 EN2 W
SH2 DO-D7 Z
DØ-D7
U30 8283
U31 8287
STB
OE


Vcc
U22
74S197
b7
b6
b5
b4
CLR
CLK2
CLK1
GND
½ U23
BP
PGS
PR
CLK
U24
b3
b2
b1
b0
SH7 PDS
BM
SH3 CKφ
BC
SH4 EOC
BI
¼ U27 74S08
BR
PGP
½ U18
BT
PGP
SH2 STB3
T
U25
8283
D7
D0
STB
OE
U26
8287
SH2 EN3
U
SH2 D0-D7
Z
D0-D7


12V
RP1
U36, U37
7406
STB A R
U32
8283
D7
D0
EN4 SH2 S
U33
8287
STB5 SH2 N
U34
8283
SH2 D0-D7 Z
S2
RESERVED
RXMIE
TXMIE
MUXCNT1
MUXCNT0
RP2
U35
8287
MUX SW1 IND2
MUX SW1 IND1
MUX SW2 IND2
MUX SW2 IND1
EN5 SH2 P


RSC-82
IF
CONTROLLER
BOARD



APPENDIX B

RDADS SOFTWARE SOURCE LISTINGS

APPENDIX B

SOFTWARE SOURCE LISTINGS

```
ROMAIN: DO;

/*
|------------------------------------------------------------|
|------------------------------------------------------------|
|                                                            |
|                      REMOTE OPERATIONS                     |
|                         MAIN MODULE                        |
|                                                            |
|------------------------------------------------------------|
|------------------------------------------------------------|
*/

$INCLUDE (:F2:RMXGEN.INC)
$INCLUDE (:F1:ROCTIP.EXT)
$INCLUDE (:F1:RODFAT.EXT)
$INCLUDE (:F1:HEDCON.EXT)
$INCLUDE (:F1:IFIODR.EXT)
$INCLUDE (:F1:DISPLA.EXT)
$INCLUDE (:F1:AIHTSK.ELT)
$INCLUDE (:F1:AIREQX.EXT)

/*************************************************************/
/*                                                           */
/*                        DECLARATIONS                       */
/*                                                           */
/*************************************************************/

DECLARE (BAND, HEAD, TXP$CHAR, RXP$CHAR,
         TXPOL$NUM, RXPOL$NUM, SDL$MODE,
         RDR$MODE, TEMP$NUM, INSTR, SCRAP,
         IFC$CONTROL, GAIN$VAL) BYTE PUBLIC;
DECLARE (RADAR$RET, ELEVATION, WINDOW$BOT,
         WINDOW$TOP, TEMP$WORD, RDR$RET) ADDRESS PUBLIC;
DECLARE (GATE$VAL, RANGE$VAL) (2) BYTE PUBLIC;
DECLARE PULSBM ADDRESS EXTERNAL;

DECLARE RDR$MSG STRUCTURE(
```

```
        MSG$HDR,
        REPS    BYTE);

DECLARE REPS BYTE PUBLIC AT (.RDR$MSG.REPS);

DECLARE (LOCALX, RXPOLX,  TXPOLX,
         MODEX,  RANGEX,  GATEX,
         HEADX,  RDR$TSX, GAINX,
         RDR$RSX, IFAIRX) EXCHANGE$DESCRIPTOR PUBLIC;

DECLARE IFAIRM AIHMSG$DESCRIPTOR PUBLIC;
DECLARE RQL2EX INT$EXCHANGE$DESCRIPTOR PUBLIC;
DECLARE RQCLID TASK$DESCRIPTOR EXTERNAL;

DECLARE POLAR (2) BYTE DATA('HV');
DECLARE TXC     LITERALLY '0';
DECLARE RXC     LITERALLY '1';
DECLARE CAL     LITERALLY '0';
DECLARE OPR     LITERALLY '1';
DECLARE SNGL    LITERALLY '1';
DECLARE CONT    LITERALLY '2';
DECLARE DA$CHAN LITERALLY '6'; /*Digital Attenuator */
DECLARE TP$CHAN LITERALLY '3'; /*Transmit Pulse */
DECLARE CS$CHAN LITERALLY '2'; /*Control and Status*/
DECLARE RD$CHAN LITERALLY '4'; /*Range Delay*/
DECLARE GP$CHAN LITERALLY '5'; /*Gate Pulse*/
DECLARE MX$CHAN LITERALLY '7'; /*Multiplexer*/
DECLARE IFC$SRI LITERALLY '2'; /*IFC Sevice Req. Intr. Level*/
DECLARE RD$CAL  LITERALLY '38H'; /*Range Delay for Cal.*/
DECLARE GP$CAL  LITERALLY '7BH'; /*Gate Pulse for Cal.*/
DECLARE LOCAL   LITERALLY '0'; /*SDL Mode Flag for Local*/
DECLARE REMOTE  LITERALLY '0FFH'; /* SDL Mode for Remote*/


/*********************************************************/
/*                                                       */
/*                    FUNCTION ROUTINES                  */
/*                                                       */
/*********************************************************/


/************************ POL$CTI ************************/

        POL$CTI: PROCEDURE BYTE;
/*_____________________________________________________

                                                      */
_____________________________________________________*/
```

```
        CALL RST$CCLIB;
        IF SPOTEQ THEN
                RETURN ID$POL;
        IF SPOTQM THEN
                RETURN 2;
        RETURN OFFH;

END POL$CTI;
/************************ NUM$REC ************************/

        NUM$REC: PROCEDURE (RET$VAL$PNTR, MAX$VAL) BYTE;
/*_____________________________________________________________

Recognize a number specification or query.

Scans the current command line for a valid number
specification that must start with an equal sign.
Check for valid number is done using SCAN$NUM.

Return:
        0 - Valid Spec -> RTRN$VAL changed.
        1 - Valid Query
        80 - Invalid.

_____________________________________________________________*/

        DECLARE (RET$VAL$PNTR, NEW$NUM, MAX$VAL) ADDRESS;
        DECLARE RET$VAL BASED RET$VAL$PNTR ADDRESS;

        IF SPOTEQ THEN
          DO;
            IF SCAN$NUM(.NEW$NUM) THEN
              DO;
                IF NEW$NUM <= MAX$VAL THEN
                  DO;
                    RET$VAL = NEW$NUM;
                    RETURN 0;
                  END;
              END;
            RETURN 80H;
          END;
        ELSE IF SPOTQM THEN
            RETURN 1;
        ELSE
            RETURN 80H;
END NUM$REC;

/************************ DS$CTI ************************/

        DS$CTI: PROCEDURE (RET$VAL$PNTR) BYTE;
```

```
/*
Delay Specification CTI.

This routine checks for a valid delay specification in the
current command line at the current position.

RTRN$VALPNTR:
        Valid spec. returned to this address.
Return:
        0 - Normal Valid Spec.
        1 - Normal Valid Query
        2 - Valid Calibration Spec.
        3 - Valid Calibration Query
        80H or 82H - Invalid Tail

                                                        */

        DECLARE (RET$VAL$PNTR, NEW$NUM) ADDRESS;
        DECLARE RET$VAL BASED RET$VAL$PNTR ADDRESS;
        DECLARE (RET$INST, KWS$PRESENT) BYTE;
        DECLARE VAL$KWB (*) BYTE DATA('VALUE');

        KWS$PRESENT = FALSE;
        CALL RST$CCLIB;
        IF REC$CAL THEN
            KWS$PRESENT = TRUE;
        ELSE
            SCRAP = REC$OP;
        RET$INST = NUM$REC(RET$VAL$PNTR, OFFH);
        IF KWS$PRESENT THEN
            RETURN  (RET$INST + 2);
        RETURN  RET$INST;

END DS$CTI;


/************************ IFGAIN ************************/

        IFGAIN:  PROCEDURE (GAIN) BYTE;
/*
        IF Gain setting procedure.

        Form of call:
                        flag = IFGAIN(gain$val)
        where,
                gain$val is a byte value for the desired
                IF Receiver gain in absolute hex (0=min.
                to FF=max).
        and
```

```
                flag is a value returned to indicate the
                success of the attempted communication to
                the IFC.

        NOTE:  There are currently only 7 bits used to
        control the gain making 7F the actual max gain.
        The flag is returned TRUE (0FFH) to signal failure
        and FALSE (0) to indicate succes.
                                                          */
------------------------------------------------------------
        DECLARE GAIN BYTE;

                CALL IFWRIT(DA$CHAN,GAIN);
                IF GAIN<>IFREAD(DA$CHAN)
                        THEN RETURN 0FFH;
                        ELSE RETURN 0;
        END IFGAIN;

/************************ RDRSIG ************************/

        RDRSIG:  PROCEDURE ADDRESS;
/*------------------------------------------------------------

        RDRSIG is a procedure for reading the Analog
        Radar Return signal via the analog input task
        AIHTSK.

        Form of call:
                        return$val = RDRSIG;
        where;
                return$val is the value of the returned
                signal in signed binary ( not 2's
                compliment!).

                                                          */
------------------------------------------------------------
/*      --  Wait for IFC Service request interrupt --     */
            SCRAP = RQWAIT(.RQL2EX,0);
/*      --  Flash 80/24 Diagnostic Lamp to indicate
            operation.                            --      */
            OUTPUT(0D6H) = 0;
/*      --  Send request for Analog Signal        --      */
            CALL RQSEND(.AIREQX,.IFAIRM);
/*      --  Wait for response from AIHTSK.        --      */
            SCRAP = RQWAIT(.IFAIRX,0);
/*      --  Return the aquired value              --      */
            RETURN RDR$RET;
        END RDR$SIG;

/********************** RADAR$ERROR **********************/

RADAR$ERROR:  PROCEDURE (ERROR$CHAR);
```

```
/*
Displays the radar error character (number) passed
followed by the message " - RADAR ERROR<CR><LF>".

                                                        */

        DECLARE ERROR$CHAR BYTE;

        CALL DISPLAY(.ERROR$CHAR,1);
        CALL DISPLAY(.(' - RADAR ERROR',0DH,0AH),16);
        RETURN;

END RADAR$ERROR;

/********************* AVG$RDR$SIG **********************/

        AVG$RDR$SIG: PROCEDURE ADDRESS;
/*
        This routine skips 1 radar return value and
        accumulates the next 8 values.  The routine
        will return the integer sum of the 8 values.

                                                        */

        DECLARE SUM$RDR$SIG ADDRESS;
        DECLARE I BYTE;

/*      --  Skip the first two returns to insure gain is
                set for the entire sample.      --      */
        SCRAP = RQWAIT(.RQL2EX,0);
        SCRAP = RQWAIT(.RQL2EX,0);

/*      --  Reset the signal total to 0.        --      */
        SUM$RDR$SIG = 0;

/*      -- Accumulate 8 RDR$SIG return values.  --      */
        DO I = 1 TO 8;
                SUM$RDR$SIG = SUM$RDR$SIG + RDR$SIG;
        END;


/*      --  Return the total.   --      */
        RETURN SUM$RDR$SIG;

END AVG$RDR$SIG;

/************************* DAGC *************************/
```

```
        DAGC:   PROCEDURE BYTE;
/*______________________________________________________________

        Digital Automatic Gain Control Implementation
        Routine.

        Form of call:
                        gain$val=DAGC;
        where,
                gain$val is the value of the gain setting
                that placed the IF analog output level
                into the acceptable window.

______________________________________________________________*/

        DECLARE READ$TIMES BYTE;
        DECLARE FINE$THRESH ADDRESS DATA(2338H); /*-2.4V*/
        DECLARE RUF$THRESH ADDRESS DATA(3D40H);  /*-0.4V*/
        DECLARE (GAIN,I,SEEK) BYTE;
        DECLARE NEXT$BIT BYTE;

/*      --  Initialize data.                    --      */
        READ$TIMES = 0;
        GAIN = 0;
        SCRAP = IFGAIN(GAIN);
        RADAR$RET = AVG$RDR$SIG;
        DO WHILE RADAR$RET > RUF$THRESH AND READ$TIMES < 16;
/*      --  Increment to rough threshold.       --      */
                GAIN = GAIN + 8;  /* Corresponds to an
                                     increment of 4dB.  */
/*      --  Output gain setting to IFC          --      */
                SCRAP = IFGAIN(GAIN);
/*      --  Get the Analog return value from the IFC. --*/
                RADAR$RET = AVG$RDR$SIG;
                READ$TIMES = READ$TIMES + 1;
        END;
        IF RADAR$RET > RUF$THRESH THEN
                CALL RADAR$ERROR('1');
        DO WHILE RADAR$RET > FINE$THRESH AND READ$TIMES < 26;
                GAIN = GAIN + 1; /* Corresponds to an
                                    increment of 1/2dB. */
                SCRAP = IFGAIN(GAIN);
                RADAR$RET = AVG$RDR$SIG;
                READ$TIMES = READ$TIMES + 1;
        END;
        IF RADAR$RET < FINE$THRESH THEN
                DO WHILE READ$TIMES < 26;
                        SCRAP = AVG$RDR$SIG;
                        READ$TIMES = READ$TIMES + 1;
                END;
```

```
        ELSE CALL RADAR$ERROR('2');
/*      -- Reset gain in IF to zero to guard against
           saturation.  The previous setting may have
           saturated the amps.                    --      */
               SCRAP = IFGAIN(0);
               SCRAP = RDR$SIG;  /* Scrap analog value,
                                    allow to settle.   */
        IF GAIN > 7FH THEN
               GAIN = 7FH;

        RETURN GAIN;

        END DAGC;



/*********************************************************/
/*                                                       */
/*                    COMMAND ROUTINES                   */
/*                                                       */
/*********************************************************/

/************************ LOCAL ************************/

        LOCAL$CMD: PROCEDURE PUBLIC;
/*_______________________________________________________

This command will be used to put the Remote Operations
Utility into the Local (CRT Present) Mode of operation.

_______________________________________________________*/

        CALL INITDSP;
        SDL$MODE = REMOTE;
        DO FOREVER;
               CLI$MSG$PNTR = RQWAIT(.LOCALX,0);
               CALL RST$CCLIB;
               IF SEARCH(.('OFF'),3)
                 AND CCLIB$MT THEN
                   DO;
                       SDL$MODE = REMOTE;
                       CALL DISPLAY(.('REMOTE',CR,LF),8);
                   END;
               ELSE IF CCLIB$MT OR
                 (SEARCH(.('ON'),2) AND CCLIB$MT) THEN
                   DO;
                       SDL$MODE = LOCAL;
                       CALL INICRT;
```

```
                        CALL DSP$MODE;
                        CALL DSP$GAIN;
                        CALL DSP$RANGE;
                        CALL DSP$GATE;
                        CALL DSP$HEAD;
                        CALL DSP$POL(TXC);
                        CALL DSP$POL(RXC);
                    END;
                ELSE
                    CALL CMD$ERR;
                CALL RQSEND(CLI$MSG.RESPONSE$EXCHANGE,
                                CLI$MSG$PNTR);
        END;

END LOCAL$CMD;

/************************* TXPOL ************************/

        TXPOL$CMD:  PROCEDURE PUBLIC;
/*_____________________________________________________

TXPOL - Command used to set or check the Transmit
polarization.

_____________________________________________________*/

        CALL INIHED;
        TXP$CHAR = 'H';
        TXPOL$NUM = 0;
        CALL TXPCON(0);

        DO FOREVER;
            CLI$MSG$PNTR = RQWAIT(.TXPOLX,0);
            INSTR = POL$CTI;
            IF INSTR <> OFFH THEN
              DO;
                IF INSTR < 2 THEN
                  DO;
                    CALL TXPCON(INSTR);
                    TX$POL$NUM = INSTR;
                    TXP$CHAR = POLAR(INSTR);
                  END;
                CALL DSP$POL(TXC);
              END;
            ELSE
                CALL CMD$ERR;
            CALL RQSEND(CLI$MSG.RESPONSE$EXCHANGE,
                        CLI$MSG$PNTR);
        END;
```

```
END TXPOL$CMD;

/************************ RXPOL ************************/

        RXPOL$CMD:  PROCEDURE PUBLIC;
/*_____________________________________________________

RXPOL  -  Command used to set or check the Receiver
polarization.

_____________________________________________________*/

        RXP$CHAR = 'H';
        RXPOL$NUM = 0;
        CALL RXPCON(0);

        DO FOREVER;
            CLI$MSG$PNTR = RQWAIT(.RXPOLX,0);
            INSTR = POL$CTI;
            IF INSTR <> OFFH THEN
              DO;
                IF INSTR < 2 THEN
                  DO;
                    CALL RXPCON(INSTR);
                    RX$POL$NUM = INSTR;
                    RXP$CHAR = POLAR(INSTR);
                  END;
                CALL DSP$POL(RXC);
              END;
            ELSE
                CALL CMD$ERR;
            CALL RQSEND(CLI$MSG.RESPONSE$EXCHANGE,
                        CLI$MSG$PNTR);
        END;

END RXPOL$CMD;


/************************ MODE$CMD ************************/

        MODE$CMD: PROCEDURE PUBLIC;
/*_____________________________________________________

MODE  -  Command used to select between calibrate or
operate mode and between single or continuous operation.

_____________________________________________________*/

        DECLARE (NEW$MODE, NEW$REPS) BYTE;
```

```
RDR$MSG.LENGTH = 10;
RDR$MSG.TYPE = 140;
RDR$MSG.RESPONSE$EXCHANGE = .RDR$RSX;
RDR$MSG.REPS = SNGL;

RDR$MODE = CAL;
NEW$MODE = OFFH;
NEW$REPS = OFFH;
CALL COPCON(1,CAL);
CALL COPCON(2,CAL);
CALL COPCON(3,CAL);

DO FOREVER;
    CLI$MSG$PNTR = RQWAIT(.MODEX,0);
    CALL RST$CCLIB;
    IF SPOTEQ THEN
      DO;
        IF REC$CAL THEN
          DO;
             NEW$MODE = CAL;
             CALL PULSBM;
          END;
        ELSE IF REC$OP THEN
          DO;
             NEW$MODE = OPR;
             CALL PULSBM;
          END;
        IF REC$CONT THEN
          NEW$REPS = CONT;
        ELSE IF REC$SNGL THEN
          NEW$REPS = SNGL;
        IF (NEW$MODE = OFFH) AND
           (NEW$REPS = OFFH) THEN
                CALL CMD$ERR;
        ELSE IF CCLIB$MT THEN
          DO;
            IF NEW$MODE < OFFH THEN
              DO;
                RDR$MODE = NEW$MODE;
                NEW$MODE = OFFH;
                CALL IFWRIT(RD$CHAN,
                  NOT(RANGE$VAL(RDR$MODE)+80H));
                CALL IFWRIT(GP$CHAN,
                  NOT(GATE$VAL(RDR$MODE)+80H));
                CALL COPCON(HEAD,RDR$MODE);
              END;
            IF NEW$REPS < OFFH THEN
              DO;
                SCRAP = RQWAIT(.RDR$RSX,0);
```

```
                    REPS = NEW$REPS;
                    NEW$REPS = OFFH;
                    CALL RQSEND(.RDR$TSX,.RDR$MSG);
                  END;
                CALL DSP$MODE;
                IF SDL$MODE = LOCAL THEN
                   DO;
                      CALL DSP$GATE;
                      CALL DSP$RANGE;
                   END;
              END;
            ELSE
              CALL CMD$ERR;
          END;
        ELSE IF SPOT$QM THEN
            CALL DSP$MODE;
        ELSE
            CALL CMD$ERR;
        CALL RQSEND(CLI$MSG.RESPONSE$EXCHANGE,
                    CLI$MSG$PNTR);
    END;

END MODE$CMD;

/********************** RANGE$CMD **********************/

    RANGE$CMD: PROCEDURE PUBLIC;
/*_____________________________________________________

RANGE  -  Command used to set the calibrate and operate
values of the IFC Range Delay parameter.

_____________________________________________________*/

    DECLARE OUT$VAL BYTE;

    RANGE$VAL(OPR), RANGE$VAL(CAL) = NOT(RD$CAL) + 80H;
    CALL IFWRIT(CS$CHAN,0BH);
    CALL IFWRIT(TP$CHAN,70H);
    CALL IFWRIT(RD$CHAN,RD$CAL);

    DO FOREVER;
      CLI$MSG$PNTR = RQWAIT(.RANGEX,0);
      INSTR = DS$CTI(.TEMP$WORD);
      OUT$VAL = NOT(LOW(TEMP$WORD)+80H);
      IF INSTR < 4 THEN
        DO;
          DO CASE INSTR;
            DO;                          /* Case 0 */
              IF RDR$MODE = OPR THEN
```

```
                CALL IFWRIT(RD$CHAN,OUT$VAL);
              RANGE$VAL(OPR) = TEMP$WORD;
            END;
            ;                              /* Case 1 */
            DO;                            /* Case 2 */
              IF RDR$MODE = CAL THEN
                CALL IFWRIT(RD$CHAN,OUT$VAL);
              RANGE$VAL(CAL) = TEMP$WORD;
            END;
            ;                              /* Case 3 */
          END;
          CALL DSP$RANGE;                  /* All Cases */
        END;
      ELSE
        CALL CMD$ERR;
      CALL RQSEND(CLI$MSG.RESPONSE$EXCHANGE,
                  CLI$MSG$PNTR);
    END;

END RANGE$CMD;


/************************ GATE$CMD ************************/

    GATE$CMD: PROCEDURE PUBLIC;
/*_____________________________________________________________

GATE  -  Command used to set the calibrate and operate
values of the IFC Gate Width parameter.

                                                            */
_____________________________________________________________*/

    DECLARE OUT$VAL BYTE;

    GATE$VAL(OPR), GATE$VAL(CAL) = NOT(GP$CAL) + 80H;
    CALL IFWRIT(GP$CHAN,GP$CAL);

    DO FOREVER;
      CLI$MSG$PNTR = RQWAIT(.GATEX,0);
      INSTR = DS$CTI(.TEMP$WORD);
      OUT$VAL = NOT(LOW(TEMP$WORD)+80H);
      IF INSTR < 4 THEN
        DO;
          DO CASE INSTR;
            DO;                            /* Case 0 */
              IF RDR$MODE = OPR THEN
                CALL IFWRIT(GP$CHAN,OUT$VAL);
              GATE$VAL(OPR) = TEMP$WORD;
            END;
            ;                              /* Case 1 */
```

```
            DO;                          /* Case 2 */
              IF RDR$MODE = CAL THEN
                CALL IFWRIT(GP$CHAN,OUT$VAL);
              GATE$VAL(CAL) = TEMP$WORD;
            END;
            ;                            /* Case 3 */
          END;
          CALL DSP$GATE;                 /* All Cases */
        END;
      ELSE
        CALL CMD$ERR;
      CALL RQSEND(CLI$MSG.RESPONSE$EXCHANGE,
                  CLI$MSG$PNTR);
    END;

END GATE$CMD;


/*********************** HEAD$CMD ************************/

    HEAD$CMD: PROCEDURE PUBLIC;
/*_____________________________________________________________

HEAD or BAND  -  Commands that select what RF head will
be used.

_____________________________________________________________*/

    DECLARE MUX$XREF (*) BYTE DATA(0,1,0,2,3);
    DECLARE BAND$CHAR (*) BYTE DATA('@XLCR');

    BAND = 'X';
    HEAD = 1;
    CALL IFWRIT(MX$CHAN,MUX$XREF(HEAD));

    DO FOREVER;
      CLI$MSG$PNTR = RQWAIT(.HEADX,0);
      CALL RST$CCLIB;
      IF SPOTEQ THEN
        DO;
          TEMP$NUM = ID$HED;
          IF TEMP$NUM = OFFH THEN
            CALL CMD$ERR;
          ELSE
            DO;
              CALL COPCON(HEAD,CAL);
              HEAD = TEMP$NUM;
              CALL COPCON(HEAD,RDR$MODE);
              BAND = BAND$CHAR(TEMP$NUM);
              CALL IFWRIT(MX$CHAN,MUX$XREF(HEAD));
```

```
                CALL DSP$HEAD;
              END;
          END;
        ELSE IF SPOTQM THEN
          CALL DSP$HEAD;
        ELSE
          CALL CMD$ERR;
        CALL RQSEND(CLI$MSG.RESPONSE$EXCHANGE,
                    CLI$MSG$PNTR);
      END;

END HEAD$CMD;


/************************ GAIN$CMD ************************/

      GAIN$CMD: PROCEDURE PUBLIC;
/*_____________________________________________________________

IFGAIN  -  Command used to set IF Digital attenuator (and
thereby the IF Gain) manually.

_____________________________________________________________*/

      GAIN$VAL = 0;
      CALL IFWRIT(DA$CHAN,0);

      DO FOREVER;
        CLI$MSG$PNTR = RQWAIT(.GAINX,0);
        CALL RST$CCLIB;
        INSTR = NUM$REC(.TEMP$WORD,07FH);
        IF INSTR < 2 THEN
          DO;
            IF INSTR = 0 THEN
              DO;
                GAIN$VAL = TEMP$WORD;
                SCRAP = IFGAIN(GAIN$VAL);
              END;
            CALL DSP$GAIN;
          END;
        ELSE
          CALL CMD$ERR;
        CALL RQSEND(CLI$MSG.RESPONSE$EXCHANGE,
                    CLI$MSG$PNTR);
      END;

END GAIN$CMD;

/************************ RDR$TSK ************************/
```

```
        RDR$TSK: PROCEDURE PUBLIC;
/*_____________________________________________________________

RDR$TASK  -  Free running data acquistion task invoked by
the MODE command entries.

                                                            */
_____________________________________________________________

        DECLARE RDR$REPS BYTE;
        DECLARE NEW$MSG ADDRESS;
        DECLARE PIT$CNTRL$WORD BYTE DATA(01110110B);
        DECLARE PIT$COUNT$LO BYTE DATA(15);
        DECLARE PIT$COUNT$HI BYTE DATA(0);

        RDR$MODE = CAL;
/*      -- Initialize the Analog Input Handler Request
           Message used to acquire measurement of the
           Radar return signal.                  --      */
        IFAIRM.LENGTH = 14;
        IFAIRM.TYPE = AI$SNGL$TYPE;
        IFAIRM.RESPONSE$EXCHANGE = .IFAIRX;
        IFAIRM.CHANNEL = 0;
        IFAIRM.DATA$PNTR = .RDR$RET;
        IFAIRM.COUNT = 1;
/*      -- Initialize Counter 1 of the PIT for divide by
              50 operation.  This effectively reduces
              the data acquisition rate by 50.        */
        OUTPUT(0DFH) = PIT$CNTRL$WORD;
        OUTPUT(0DDH) = PIT$COUNT$LO;
        OUTPUT(0DDH) = PIT$COUNT$HI;

        CALL RQSEND(.RDRRSX,.RDRMSG);

        DO FOREVER;
          SCRAP = RQWAIT(.RDRTSX,0);
          RDR$REPS = RDR$MSG.REPS;
          CALL RQSEND(.RDRRSX,.RDRMSG);
          CALL RQELVL(IFC$SRI);
          DO WHILE RDR$REPS = SNGL OR RDR$REPS = CONT;
            IF RDR$REPS = SNGL THEN
                RDR$REPS = 0;
            GAIN$VAL = DAGC;
            CALL DSP$GAIN;
            NEW$MSG = RQACPT(.RDRTSX);
            IF NEW$MSG > 0 THEN
              DO;
                RDR$REPS = RDR$MSG.REPS;
                CALL RQSEND(.RDRRSX,.RDRMSG);
              END;
          END;
```

```
            CALL RQDLVL(IFC$SRI);
          END;

END RDR$TSK;


END ROMAIN;      /* End of module. */

EOF
```

```
ROCTIP:  DO;

/*
=============================================================

                    REMOTE OPERATIONS
            COMMAND TAIL INTERPRETATION PACKAGE

=============================================================
*/
$INCLUDE (:F2:RMXGEN.INC)

/*************************************************************/
/*                                                           */
/*              GLOBAL  DECLARATIONS                         */
/*                                                           */
/*************************************************************/

DECLARE CCLIB$PNTR ADDRESS PUBLIC;
DECLARE (SCANNER, BALANCE, MARKER) BYTE PUBLIC;
DECLARE CCLIB BASED CCLIB$PNTR (117) BYTE;
DECLARE (MARK, PREV$BAL) BYTE PUBLIC;

DECLARE CLI$MSG$PNTR ADDRESS PUBLIC;
DECLARE CLI$MSG BASED CLI$MSG$PNTR STRUCTURE(
                MSG$HDR,
                COUNT     ADDRESS,
                BUFF$ADR  ADDRESS);

DECLARE NEXT$CHAR LITERALLY 'CALL SCAN';
DECLARE PREV$CHAR LITERALLY 'CALL SCAN$BAK';
DECLARE SAVE LITERALLY 'CALL SAVE$SAB';
DECLARE RESTORE LITERALLY 'CALL RESTOR$SAB';


/*************************************************************/
/*                                                           */
/*                  GLOBAL PROCEDURES                        */
/*                                                           */
/*************************************************************/


/********************** RST$CCLIB ************************/

       RST$CCLIB: PROCEDURE PUBLIC;
/*_____________________________________________________________

RST$CCLIB will reset the values of SCANNER, BALANCE, and
CCLIB$PNTR based on the current values in the CLI$MSG
structure (pointed to by CLI$MSG$PNTR).
```

```
        Form of call:
                        CALL RST$CCLIB;

mordoc

-------------------------------------------------------------*/

        SCANNER = 0;
        BALANCE = CLI$MSG.COUNT;
        CCLIB$PNTR = CLI$MSG.BUFF$ADR;
        RETURN;

END RST$CCLIB;


/************************ SAVE$SAB ************************/

        SAVE$SAB: PROCEDURE PUBLIC;
/*-------------------------------------------------------------

SAVE$SAB can be used to save the current values for
SCANNER and BALANCE in MARK and PREV$BAL respectively.

        Form of call:
                        CALL SAVE$SAB;
        or
                        SAVE;           (in this module)

-------------------------------------------------------------*/

        MARK = SCANNER;
        PREV$BAL = BALANCE;
        RETURN;

END SAVE$SAB;

/************************ RESTOR$SAB ************************/

        RESTOR$SAB: PROCEDURE PUBLIC;
/*-------------------------------------------------------------

This routine performs the inverse of SAVE$SAB by restoring
the values of SCANNER and BALANCE to the values saved in
MARK and PREV$BAL.

        Form of call:
                        CALL RESTOR$SAB;
        or
                        RESTORE;        (in this module)
```

```
------------------------------------------------------------*/

        SCANNER = MARK;
        BALANCE = PREV$BAL;
        RETURN;

END RESTOR$SAB;

/************************* SCAN *************************/

        SCAN:   PROCEDURE PUBLIC;
/*------------------------------------------------------------

SCAN will advance the position in the CCLIB by simply
incrementing SCANNER and decrementing BALANCE.

        Form of call:
                        CALL SCAN;
        or
                        NEXT$CHAR;      (in this module)

------------------------------------------------------------*/

        SCANNER = SCANNER + 1;
        BALANCE = BALANCE - 1;
        RETURN;

END SCAN;

/************************ SCAN$BAK ************************/

        SCAN$BAK: PROCEDURE PUBLIC;
/*------------------------------------------------------------

This procedure performs the inverse of SCAN (backs up one
character in the CCLIB).

        Form of call:
                        CALL SCAN$BAK;
        or
                        PREV$CHAR;      (in this module)

------------------------------------------------------------*/

        SCANNER = SCANNER - 1;
        BALANCE = BALANCE + 1;
        RETURN;

END SCAN$BAK;
```

```
/************************ SKIP ****************************/

        SKIP:   PROCEDURE BOOLEAN PUBLIC;
/*_____________________________________________________________

SKIP will advance the CCLIB pointer, SCANNER, to the first
non-blank character in the CCLIB.

        Form of call:
                        flag = SKIP;

        where,
                "flag" is a boolean (byte) variable used to
                signal that the SKIP operation encountered
                the "end-of-line" (break sequence) before
                any other non-blank character was found.

The procedure will return a true (OFFH) if successful with-
out encountering the "end-of-line" and false otherwise.
SCANNER is assumed to index the starting position in CCLIB
at entry and will be set to index the non-blank character
at exit.  BALANCE is decremented by the number of spaces
skipped.  MARKER is not changed.
_____________________________________________________________*/

        IF BALANCE = 0 THEN     /* Important Step ! ! ! */
                RETURN FALSE;   /* Do Not Delete ! ! !  */
        DO WHILE BALANCE > 0
                 AND CCLIB(SCANNER) = ' ';
                        NEXT$CHAR;
          END;
        IF CCLIB(SCANNER) = CR
            OR BALANCE = 0
                THEN    RETURN FALSE;
                ELSE    RETURN TRUE;

END SKIP;


/********************** CCLIB$MT **********************/

        CCLIB$MT: PROCEDURE BOOLEAN PUBLIC;
/*_____________________________________________________________

CCLIB$MT checks the CCLIB to see if it is empty.

        Form of call:
                        flag = CCLIB$MT;
        where,
```

```
                "flag" is a boolean variable set true if
                the CCLIB is empty or false otherwise.

The CCLIB is considered to be empty if BALANCE is zero or
the break sequence (carriage return) is the next non-blank
character in the buffer.  The effects of CCLIB$MT on the
position in CCLIB (values for SCANNER and BALANCE) are the
same as for SKIP (SKIP is used in the determination).

-------------------------------------------------------------*/

        RETURN NOT(SKIP);

END CCLIB$MT;


/********************** VALID$DEL ************************/

        VALID$DEL: PROCEDURE BOOLEAN PUBLIC;
/*-------------------------------------------------------------

This routine will check the next character in the CCLIB to
see if it is a valid delimeter (blank or carriage return).

        Form of call:
                        flag = VALID$DEL;
        where,
                "flag" is a boolean flag set to indicate
                success (true or OFFH) or failure (false
                or 0).

The globals SCANNER, BALANCE, and MARKER are not affected
by VALID$DEL.

-------------------------------------------------------------*/

        IF CCLIB(SCANNER) = ' '
           OR CCLIB(SCANNER) = CR
                THEN RETURN TRUE;
                ELSE RETURN FALSE;

END VALID$DEL;


/************************ SEARCH ************************/

        SEARCH: PROCEDURE (PNTR,LEN) BOOLEAN PUBLIC;
/*-------------------------------------------------------------

SEARCH will scan the CCLIB for a specified string.
```

```
Form of call:
                flag = SEARCH(pntr,len);
where,
        "pntr" is an address pointer to the string

        "len" is a byte value of the number of
        characters in the string
and
        "flag" is a boolean flag that signals the
        success or failure of the operation.

SEARCH will return a true if the operation is a success
and a false if it is a failure.  The routine signals a
success if the first non-blank character string matches
the string specified or an abbreviation of the string.
An abbreviation is one or more characters of the string
followed by a space or carriage return (CR).  A failure is
signaled if the matched string is not terminated with a
blank or CR or a match is not found.  SCANNER is assumed to
index the starting position at entry to this procedure.
If successful SCANNER is set to point to the terminating
character.  If unsuccessful SCANNER is set to point to the
first non-blank character it encounters.  BALANCE is set
to reflect the change in position in CCLIB and MARKER is
not changed.

-------------------------------------------------------------*/

DECLARE (I, LEN) BYTE;
DECLARE PNTR ADDRESS;
DECLARE STRING BASED PNTR (1) BYTE;

I = 0;
IF SKIP THEN
     DO;
        SAVE;
        DO WHILE CCLIB(SCANNER) = STRING(I)
             AND I < LEN
             AND BALANCE > 0;
                I = I + 1;
                NEXT$CHAR;
        END;
        IF I = 0 THEN
                RETURN FALSE;
        IF VALID$DEL
          AND  BALANCE > 0
           THEN
                RETURN TRUE;
        RESTORE;
```

```
            END;
        RETURN FALSE;

END SEARCH;

/************************ SPOTEQ ************************/

        SPOTEQ: PROCEDURE BOOLEAN PUBLIC;
/*_____________________________________________________

SPOTEQ will check the CCLIB for the occurrence of an equal
sign ignoring leading blanks.

        Form of call:
                        flag = SPOTEQ;
        where,
                "flag" is a boolean value set true if the
                operation is a success and false if it is
                a failure.

SCANNER is first set to skip leading blanks (via SKIP).
If the next character is an equals sign it is set to point
to the character just after it.  Otherwise it will point
to the non-blank character.  BALANCE is adjusted to show
this change of position in the CCLIB.  MARKER remains
unchanged.

_____________________________________________________*/

        IF SKIP THEN
            DO;
                IF CCLIB(SCANNER) = '=' THEN
                    DO;
                        NEXT$CHAR;
                        RETURN TRUE;
                    END;
            END;
        RETURN FALSE;

END SPOTEQ;

/************************ SPOTQM ************************/

        SPOTQM: PROCEDURE BOOLEAN PUBLIC;
/*_____________________________________________________

SPOTQM will check CCLIB for the occurrence of a question
mark ignoring leading blanks.

        Form of call:
```

```
                        flag = SPOTQM;
        where,
                "flag" is a boolean value set true if the
                operation is a success and false if it is
                a failure.

SCANNER is first set to skip leading blanks (via SKIP).
If the next character is a question mark and is the last
logical character then SCANNER is set to index the break
sequence and the routine returns OFFH (true).  Otherwise
SCANNER will point to the first non-blank character found
and 0 (false) is returned.  BALANCE is adjusted to show
this change of position in the CCLIB.  MARKER remains
unchanged.

-------------------------------------------------------------*/

        IF SKIP THEN
           DO;
                IF CCLIB(SCANNER) = '?' THEN
                    DO;
                        SAVE;
                        NEXT$CHAR;
                        IF CCLIB$MT THEN
                            RETURN TRUE;
                        RESTORE;
                    END;
                RETURN FALSE;
            END;
        RETURN TRUE;

END SPOTQM;

/********************** SCAN$NUM **********************/

        SCAN$NUM: PROCEDURE (PNTR) BOOLEAN PUBLIC;
/*-------------------------------------------------------------

The SCAN$NUM routine will scan the CCLIB for an ASCII
string representing a constant value.

        Form of call:
                        flag = SCAN$NUM(num$addr);
        where,
                "num$addr" is an address pointer to an
                address scaler where the resulting value,
                if obtained, is to be stored.
        and
                "flag" is a boolean value set to indicate
                the success of the operation (TRUE) or the
```

failure of it (FALSE).

If "flag" is returned FALSE then the contents of the address scaler pointed to by "num$addr" is undefined.

The constant value can be entered as either decimal, hex, octal, or binary. All entries require a radix descriptor except for decimal values for which the radix is optional. In other words entries without a descriptor are assumed to be decimal. The following show the characters used as the radix descriptors for each base and the maximum allowable entry.

Base	Radix Descriptor	Maximum Entry*
Decimal	"T"	65536D
Hex	"H"	FFFFH
Octal	"Q"	1777770
Binary	"Y"	1111111111111111B

* All maximum values equate to the same numerical limit.

When the routine is successful SCANNER and BALANCE are adjusted so that the CCLIB position is at the delimiting character that terminated the string. Otherwise it will be positioned to the first non-blank character found. Note also that MARKER is changed by this routine.

```
-------------------------------------------------------------*/

    DECLARE PNTR ADDRESS;
    DECLARE NUMBER BASED PNTR ADDRESS;
    DECLARE DIGITS (*) BYTE DATA('0123456789ABCDEF');
    DECLARE (NEW$NUM, OLD$NUM) ADDRESS;
    DECLARE (RADIX, I, CHAR) BYTE;


    IF SKIP THEN
        DO;                         /* Found a string. */
            SAVE;
            MARKER = SCANNER;
            CHAR = CCLIB(SCANNER);
            DO WHILE ( ( CHAR >= '0' AND CHAR <= '9' )
                       OR ( CHAR >= 'A' AND CHAR <= 'F' ) )
                     AND  BALANCE > 0;
                    NEXT$CHAR;
                    CHAR = CCLIB(SCANNER);
            END;
            IF CHAR = 'H' THEN
                    RADIX = 16;
            ELSE IF CHAR = 'Q' THEN
```

```
                RADIX = 8;
            ELSE IF CHAR = 'Y' THEN
                RADIX = 2;
            ELSE IF CHAR = 'T' OR VALID$DEL THEN
                RADIX = 10;
            ELSE                        /* Bad character */
                DO;
                    RESTORE;
                    RETURN FALSE;
                END;
            NEW$NUM, OLD$NUM = 0;
            DO WHILE MARKER < SCANNER;
                I = 0;
                DO WHILE CCLIB(MARKER) <> DIGITS(I);
                    I = I+1;
                END;
                IF I >= RADIX THEN
                    DO;
                        RESTORE;
                        RETURN FALSE;
                    END;
                NEW$NUM = OLD$NUM*RADIX + I;
                IF NEW$NUM < OLD$NUM THEN
                    DO;
                        RESTORE;
                        RETURN FALSE;
                    END;
                OLD$NUM = NEW$NUM;
                MARKER = MARKER + 1;
            END;
            NUMBER = NEW$NUM;
            IF RADIX <> 10 OR CHAR = 'T' THEN
                    NEXT$CHAR;
            IF VALID$DEL THEN
                    RETURN TRUE;
            RESTORE;
        END;
    RETURN FALSE;

END SCAN$NUM;

/************************ ID$HED ************************/

    ID$HED: PROCEDURE BYTE PUBLIC;
/*_____________________________________________________

This routine will try to identify the next non-blank
character in the CCLIB as a head specifier.

    Form of call:
```

```
                    hed$num = ID$HED;
        where,
                "hed$num" is a byte value set to indicate
                the appropriate head number or lack of one.

ID$HED will first scan CCLIB for the first non-blank
character in it.  That character is then compared to the
permissible head specifiers for a match.  Failure to find
a non-blank character or failure to find a valid match
that is properly terminated returns OFFH.  A proper match
will return the following values:

        Specifiing Character              Return Value

                X or 1  - - - - - - - - - - -   1
                L or 2  - - - - - - - - - - -   2
                C or 3  - - - - - - - - - - -   3
                R or 4  - - - - - - - - - - -   4

-------------------------------------------------------*/

        DECLARE VALID$CHARS (8) BYTE DATA('X1L2C3R4');
        DECLARE I BYTE;

        IF SKIP THEN
            DO;
                I = 0;
                DO WHILE CCLIB(SCANNER) <> VALID$CHARS(I)
                     AND I < 8;
                        I = I + 1;
                END;
                IF I < 8 THEN
                    DO;
                        NEXT$CHAR;
                        IF CCLIB$MT THEN
                                RETURN (I/2)+1;
                    END;
            END;
        RETURN OFFH;

END ID$HED;


/************************ REC$CAL ************************/

        REC$CAL:  PROCEDURE BOOLEAN PUBLIC;
/*-------------------------------------------------------

REC$CAL will recognize the presence of the keyword
"CALIBRATE" in the CCLIB.
```

```
        Form of call:
                        flag = REC$CAL;
        where,
                "flag" is a boolean value set to indicate
                the presence of the keyword (true) or the
                absence of it (false).

As usual if the keyword is found and is properly terminated
SCANNER will index the delimiter at completion.  Otherwise
it will index the first non-blank character found.  Also
BALANCE is properly adjusted and MARKER remains unchanged.

-------------------------------------------------------------*/

        DECLARE KEY$WORD$BUF (*) BYTE DATA('CALIBRATE');

        RETURN SEARCH(.KEY$WORD$BUF,LENGTH(KEY$WORD$BUF));

END REC$CAL;

/************************ REC$OP ************************/

        REC$OP: PROCEDURE BOOLEAN PUBLIC;
/*-------------------------------------------------------------

REC$OP will recognize the presence of the keyword
"OPERATE" in the CCLIB.

        Form of call:
                        flag = REC$OP;
        where,
                "flag" is a boolean value set to indicate
                the presence of the keyword (true) or the
                absence of it (false).

As usual if the keyword is found and is properly terminated
SCANNER will index the delimiter at completion.  Otherwise
it will index the first non-blank character found.  Also
BALANCE is properly adjusted and MARKER remains unchanged.

-------------------------------------------------------------*/

        DECLARE KEY$WORD$BUF (*) BYTE DATA('OPERATE');

        RETURN SEARCH(.KEY$WORD$BUF,LENGTH(KEY$WORD$BUF));

END REC$OP;

/************************ REC$CONT ************************/
```

```
        REC$CONT: PROCEDURE BOOLEAN PUBLIC;
/*_____________________________________________________________

This routine will recognize the presence of the keyword
"CONTINUOUS" in the CCLIB.

(For details see REC$CAL above.)
_____________________________________________________________*/

        DECLARE KEY$WORD$BUF (*) BYTE DATA('CONTINUOUS');

        RETURN SEARCH(.KEY$WORD$BUF,LENGTH(KEY$WORD$BUF));

END REC$CONT;


/*********************** REC$SNGL ***********************/

        REC$SNGL: PROCEDURE BOOLEAN PUBLIC;
/*_____________________________________________________________

This routine will recognize the presence of the keyword
"SINGLE" in the CCLIB.

(For details see REC$CAL above.)
_____________________________________________________________*/

        DECLARE KEY$WORD$BUF (*) BYTE DATA('SINGLE');

        RETURN SEARCH(.KEY$WORD$BUF,LENGTH(KEY$WORD$BUF));

END REC$SNGL;


/************************ ID$POL ************************/

        ID$POL: PROCEDURE BYTE PUBLIC;
/*_____________________________________________________________

This procedure can be used to identify an optional polar-
ization specifier in the CCLIB.

        Form of call:
                        pol$spec = ID$POL;
        where,
                "pol$spec" is a byte variable set to
                indicate the presence and type of the
                polarization specifier.
```

```
ID$POL will return a value of 0 if the next non-blank
character in the CCLIB is an "H" and a 1 if the character
is a "V".  Either character must be properly terminated
(VALID$DEL).  The absence of both characters or lack of
proper termination returns a value of OFFH.  When 0 or 1
is returned SCANNER is set to index the delimeter other-
wise it is set to index the first non-blank character or
break sequence.  BALANCE is set to reflect the adjustment
of position in the CCLIB.  MARKER remains unchanged.

-------------------------------------------------------------*/

        DECLARE POL$NUM BYTE;

        IF SKIP THEN
            DO;
                IF CCLIB(SCANNER) = 'H'
                  THEN  POL$NUM = 0;
                ELSE IF CCLIB(SCANNER) = 'V'
                  THEN  POL$NUM = 1;
                ELSE RETURN OFFH;
                NEXT$CHAR;
                IF VALID$DEL
                  THEN RETURN POL$NUM;
                PREV$CHAR;
            END;
        RETURN OFFH;

END ID$POL;


/*=============================================================*/

END ROCTIP;     /* End of module. */

EOF
```

```
RODFAT:  DO;

/*
=========================================================

                    REMOTE OPERATIONS
              DATA FORMATTER AND TRANSMISSION PACKAGE

=========================================================
```

This package provides routines for the formatting and display of character strings, numeric values, and other types of various structures such as lines and boxes.

There is a basic logical concept common to most all of the procedures included in this package. The output of data to the CRT (assumed to be a VT-100, CIT-101, or equ.) is accomplished in two steps. The first step is to fill a data buffer with various combinations of character strings and control sequences that are needed to accomplish the desired display operation. This buffer is then output to the console via the DISPLAY routine found in SYSUTL.LIB. Many of the following procedures perform the function of placing some type of ASCII data in the display buffer, DSP$BUF, and that is all. There are routines provided that will actually send the contents of the display buffer to the console. Most of the procedures perform very simple yet important and useful tasks. The following description of the package's data structures and the descriptions that accompany the routines should provide all the required information for using this package.

\- * - * - * - * - * - * - *

DATA STRUCTURES AND KEY DATA VARIABLES.

DSP$BUF - Display Buffer used to hold ASCII data as it is formatted and assembled for output to the system console device. This is a byte array with a maximum capacity of 256 characters.

NDEX - A byte value used to index the current position in DSP$BUF. DSP$BUF(NDEX)

refers to the current character or byte of DSP$BUF. NOTE: In this package the string "CHAR" is literally equivalent to the string "DSP$BUF(NDEX)" and is used in its place.

NUM - A byte value used by some routines to hold hex or decimal values that are converted to ASCII and stored in DSP$BUF.

NUM$3 - An address value used by routines for the same purpose as NUM but for a larger range of values.

COOR$STR - This is a data structure with three (3) members:

ROW - byte,
COL - byte,
ATTRIB - byte.

The structure is used to hold the row and column coordinates of a position on the CRT screen in ROW and COL respectively. It also holds a cursor attribute in ATTRIB.

BOX$SPEC$STR - This is a data structure with five (5) members:

ROW - byte,
COL - byte,
ATTRIB - byte,
LEN - byte,
HGT - byte.

This structure is used to hold data for specifing the location (ROW and COL), display attribute (ATTRIB), length (LEN), and hieght (HGT) of a box to be displayed on the console. The coordinate location is assumed to be that of the top-left corner of the box. The length and height are inside dimensions of the box (ie. do not include the box lines) and the overall dimensions are two greater for each value.

DATA$COOR$STR - This data structure has two (2) members:

ROW - byte,
COL - byte.

This is used to hold row and column coordinates for data placement on the screen.

LABEL$STR - This structure has four (4) members:

```
                                ROW - byte,
                                COL - byte,
                                COUNT - byte,
                                PNTR - address.
                        This structure is used to hold data for
                        the placement of a character string at a
                        location on the screen specified by ROW
                        and COL.  The begining of the string is
                        pointed to in memory by PNTR and is COUNT
                        characters long.


=============================================================
*/

$INCLUDE (:F1:ASCLO.EXT)
$INCLUDE (:F1:ASCHI.EXT)
$INCLUDE (:F1:ASCII4.EXT)
$INCLUDE (:F1:ASCII2.EXT)
$INCLUDE (:F1:BLANK.EXT)
$INCLUDE (:F1:DISPLA.EXT)
$INCLUDE (:F1:B2D.EXT)

/*************************************************************/
/*                                                           */
/*                    GLOBAL DECLARATIONS                    */
/*                                                           */
/*************************************************************/

DECLARE (TXP$CHAR,RXP$CHAR,RDR$MODE,GAIN$VAL,
         HEAD,BAND,REPS,SDL$MODE) BYTE EXTERNAL;
DECLARE (RANGE$VAL,GATE$VAL) (2) BYTE EXTERNAL;

DECLARE VALDAT ADDRESS EXTERNAL;
DECLARE TRUE LITERALLY '0FFH';
DECLARE FALSE LITERALLY '00H';
DECLARE BOOLEAN LITERALLY 'BYTE';
DECLARE FOREVER LITERALLY 'WHILE 1';
DECLARE LOCAL LITERALLY '0';
/* SPECIAL ASCII CHARACTERS */
DECLARE
  NULL          LITERALLY '00H',
  CONTROL$C     LITERALLY '03H',
  CONTROL$E     LITERALLY '05H',
  BELL          LITERALLY '07H',
  BS            LITERALLY '08H',
  TAB           LITERALLY '09H',
  LF            LITERALLY '0AH',
  VT            LITERALLY '0BH',
  FF            LITERALLY '0CH',
```

```
   CR             LITERALLY '0DH',
   CONTROL$P      LITERALLY '10H',
   CONTROL$Q      LITERALLY '11H',
   CONTROL$R      LITERALLY '12H',
   CONTROL$S      LITERALLY '13H',
   CONTROL$X      LITERALLY '18H',
   CONTROL$Z      LITERALLY '1AH',
   ESC            LITERALLY '1BH',
   QUOTE          LITERALLY '22H',
   SPACE          LITERALLY '20H',
   LCA            LITERALLY '61H',
   LCZ            LITERALLY '7AH',
   RUBOUT         LITERALLY '7FH';

DECLARE CAL       LITERALLY '0';
DECLARE OPR       LITERALLY '1';
DECLARE SNGL      LITERALLY '1';
DECLARE CONT      LITERALLY '2';
DECLARE TXC       LITERALLY '0';
DECLARE RXC       LITERALLY '1';

DECLARE DSP$BUF (256) BYTE;
DECLARE (NDEX, NUM) BYTE;
DECLARE (NUM$3) ADDRESS;

/*-------------------------------------------------------*/
/*
   |-------------------------------------------------|
   |                                                 |
   |       * * * > W A R N I N G < * * *             |
   |                                                 |
   |  This package makes frequent use of liter-      |
   |  ally declared  data strings that may make      |
   |  the programs hard to read.  The reader is      |
   |  advised to become  familiar with the fol-      |
   |  lowing in order to avoid confusion.            |
   |                                                 |
*/ |-------------------------------------------------|

DECLARE CHAR      LITERALLY 'DSP$BUF(NDEX)';
DECLARE NEXT      LITERALLY 'CALL NXNDEX';
DECLARE STUFF     LITERALLY 'CALL PUT$CHAR';
DECLARE ESCP      LITERALLY 'CALL ESCAPE';
DECLARE INTRO     LITERALLY 'CALL CSI';
DECLARE DLMT      LITERALLY 'CALL DLIMIT';
DECLARE CRNR      LITERALLY 'CALL CORNER';
DECLARE HLINE     LITERALLY 'CALL HOR$LINE';
DECLARE VLINE     LITERALLY 'CALL VER$LINE';
DECLARE CMPLT     LITERALLY 'CALL COMPLETE';
DECLARE JUMP      LITERALLY 'CALL JUMP$UP';
DECLARE LFBS      LITERALLY 'CALL DLDIAG';
```

```
/*-------------------------------------------------------*/

DECLARE VL      LITERALLY '170Q';  /* Vertical Bar Line */
DECLARE HL      LITERALLY '161Q';  /* Horizontal Line */

DECLARE NORM    LITERALLY '0';     /* Normal Character Set */
DECLARE ASCII   LITERALLY '1';     /* ASCII Character Set */
DECLARE GRAPH   LITERALLY '2';     /* Graphics Character Set */
DECLARE ALTER   LITERALLY '3';     /* Alternate Character Set */
DECLARE SPEC    LITERALLY '4';     /* Special Graphics Set */

/*      CORNER and Line Intersection Specifiers.

                2----8----1
                |    |    |
                5----4----6
                |    |    |
                3----7----0
*/
DECLARE LR      LITERALLY '0';  /* Lower Right    */
DECLARE UR      LITERALLY '1';  /* Upper Right    */
DECLARE UL      LITERALLY '2';  /* Upper Left     */
DECLARE LL      LITERALLY '3';  /* Lower Left     */
DECLARE XL      LITERALLY '4';  /* Crossing Lines */
DECLARE LT      LITERALLY '5';  /* Left Tee       */
DECLARE RT      LITERALLY '6';  /* Right Tee      */
DECLARE BT      LITERALLY '7';  /* Bottom Tee     */
DECLARE TT      LITERALLY '8';  /* Top Tee        */


DECLARE COOR$STR LITERALLY 'STRUCTURE(
                ROW     BYTE,
                COL     BYTE,
                ATTR    BYTE)';

DECLARE BOX$SPEC$STR LITERALLY 'STRUCTURE(
                ROW     BYTE,
                COL     BYTE,
                ATTR    BYTE,
                LEN     BYTE,
                HGT     BYTE)';

DECLARE LABEL$STR LITERALLY 'STRUCTURE(
                ROW     BYTE,
                COL     BYTE,
                COUNT   BYTE,
                PNTR    ADDRESS)';
```

```
DECLARE DATA$COOR$STR LITERALLY 'STRUCTURE(
                ROW     BYTE,
                COL     BYTE)';

DECLARE SEP$LINE COOR$STR DATA(9,1,1);
DECLARE SCROLL$REG COOR$STR DATA(10,1,0);
DECLARE LOGO$COOR COOR$STR DATA(2,1,0);

DECLARE LOGO (*) BYTE DATA
          ('  `bfn~~~~~~~}w  `bfn~~~~~~~}w   w}~~~~~~nfb`',CR,
        LF,'  wwww gwwwxvffe   wwww    `fffe effvxwww',CR,
        LF,'   ~~~~    `bfn~}{wo ~~~~    ow{}~~~~~~~nb');

DECLARE LINE$0 (59) BYTE DATA
        ('R E M O T E',ESC,'[B',ESC,'[15DO P E R A T I O N S',
          ESC,'[B',ESC,'[16DU T I L I T Y');
DECLARE LINE$1 (6) BYTE DATA
        ('Mode =');
DECLARE LINE$2 (24) BYTE DATA
        ('IF Gain =       x 1/2 dB');
DECLARE LINE$3 (39) BYTE DATA
        ('(16.67 nS)
',ESC,'[4mOperate',ESC,'[2CCalibrate',ESC,'[0m');
DECLARE LINE$4 (32) BYTE DATA
        ('Range Delay:',ESC,'[B',ESC,'[12DGate  Width:');
DECLARE LINE$5 (43) BYTE DATA
        ('Head =     Band =      TxPol =      RxPol =');

DECLARE LINE$LBL (6) LABEL$STR DATA(
                2,56,59,.LINE$0,
                7,3,6,.LINE$1,
                7,38,24,.LINE$2,
                2,85,39,.LINE$3,
                3,84,32,.LINE$4,
                7,69,43,.LINE$5);

DECLARE CO$COOR DATA$COOR$STR DATA(7,10);
DECLARE GN$COOR DATA$COOR$STR DATA(7,48);
DECLARE RD$COOR (2) DATA$COOR$STR DATA(3,107,3,98);
DECLARE GP$COOR (2) DATA$COOR$STR DATA(4,107,4,98);
DECLARE HED$COOR DATA$COOR$STR DATA(7,76);
DECLARE BND$COOR DATA$COOR$STR DATA(7,87);
DECLARE TXP$COOR DATA$COOR$STR DATA(7,100);
DECLARE RXP$COOR DATA$COOR$STR DATA(7,113);

DECLARE BOX$SPEC (9) BOX$SPEC$STR DATA(
                1,1,1,77,3,
                1,82,0,32,3,
                6,1,0,29,1,
                6,35,0,28,1,
```

```
                6,67,0,47,1,
                4,123,0,1,1,
                3,121,1,5,3,
                2,119,0,9,5,
                1,117,1,13,7);

/*************************************************************/
/*                                                           */
/*                    GLOBAL PROCEDURES                      */
/*                                                           */
/*************************************************************/


/************************ RESTRT ************************/

        RESTRT:  PROCEDURE  PUBLIC;
/*_____________________________________________________________

Reset the index variable NDEX to 0;
_____________________________________________________________*/

        NDEX = 0;
        RETURN;

END RESTRT;

/************************ NXNDEX ************************/

        NXNDEX:  PROCEDURE  PUBLIC;
/*_____________________________________________________________

Increment the NDEX index parameter.

NOTE:  In this package "NEXT" is literally "CALL NXNDEX".
_____________________________________________________________*/

        NDEX = NDEX + 1;
        RETURN;

END NXNDEX;

/************************ PUT$CHAR ************************/

        PUT$CHAR:  PROCEDURE (USR$CHAR) PUBLIC;
/*_____________________________________________________________

Store character in the display buffer DSP$BUF.

NOTE:  In this package "STUFF" = "CALL PUT$CHAR".
```

```
------------------------------------------------------------*/

        DECLARE USR$CHAR BYTE;

        CHAR = USR$CHAR;
        NEXT;
        RETURN;

END PUT$CHAR;

/************************ ESCAPE ************************/

        ESCAPE:  PROCEDURE  PUBLIC;
/*______________________________________________________________

Place Escape character in the output display buffer.

NOTE:  In this package "ESCP" = "CALL ESCAPE".
------------------------------------------------------------*/

        STUFF(ESC);
        RETURN;

END ESCAPE;

/************************** CSI **************************/

        CSI:  PROCEDURE  PUBLIC;
/*______________________________________________________________

Place the common Control Sequence Introducer (ESC[) in
the display buffer.

NOTE:  In this package "INTRO" = "CALL CSI".
------------------------------------------------------------*/


        ESCP;
        STUFF('[');
        RETURN;

END CSI;

/************************ DLIMIT ************************/

        DLIMIT:  PROCEDURE  PUBLIC;
/*______________________________________________________________

Put the semicolon delimiter in the display buffer.
```

```
NOTE:  In this package "DLMT" = "CALL DLIMIT".
-------------------------------------------------------------*/

        STUFF(';');
        RETURN;

END DLIMIT;

/*********************** PUT$NUM$1 ***********************/

        PUT$NUM$1:  PROCEDURE  PUBLIC;
/*-------------------------------------------------------------

Place the ASCII code for the low nibble of NUM in the
display buffer.
-------------------------------------------------------------*/

        STUFF(ASCLO(NUM));
        RETURN;

END PUT$NUM$1;

/*********************** PUT$NUM$2 ***********************/

        PUT$NUM$2:  PROCEDURE  PUBLIC;
/*-------------------------------------------------------------

Put the ASCII code for the high nibble of NUM followed
by the ASCII code for the low nibble in the display
buffer.
-------------------------------------------------------------*/

        STUFF(ASCHI(NUM));
        CALL PUT$NUM$1;
        RETURN;

END PUT$NUM$2;

/*********************** PUT$NUM$3 ***********************/

        PUT$NUM$3:  PROCEDURE  PUBLIC;
/*-------------------------------------------------------------

Place the ASCII for the last three nibbles of the address
parameter NUM$3 in the display buffer in order from left
to right.

        * * * >  NOTE:  This routine destroys the
                        contents of NUM.
-------------------------------------------------------------*/
```

```
        NUM = HIGH(NUM$3);
        CALL PUT$NUM$1;
        NUM = LOW(NUM$3);
        CALL PUT$NUM$2;
        RETURN;

END PUT$NUM$3;

/************************ SAVEC ************************/

        SAVEC:  PROCEDURE  PUBLIC;
/*_____________________________________________________

Place the Control Sequence to save the current cursor
position and attributes in the display buffer.
_____________________________________________________*/

        ESCP;
        STUFF('7');
        RETURN;

END SAVEC;

/************************ RESTORC ************************/

        RESTORC:  PROCEDURE  PUBLIC;
/*_____________________________________________________

Place the control sequence for restoring the saved
cursor position and attributes in the display buffer.
_____________________________________________________*/

        ESCP;
        STUFF('8');
        RETURN;

END RESTORC;

/************************ COMPLETE ************************/

        COMPLETE:  PROCEDURE  PUBLIC;
/*_____________________________________________________

Output the display buffer via DISPLAY.

NOTE:  In this package "CMPLT" = "CALL COMPLETE".
_____________________________________________________*/

        CALL DISPLAY(.DSP$BUF,NDEX);
```

```
        RETURN;

END COMPLETE;

/************************ STRTUP ************************/

        STRTUP:  PROCEDURE PUBLIC;
/*_____________________________________________________

Makes calls to RESTRT and SAVEC for convenience.
_____________________________________________________*/

        CALL RESTRT;
        CALL SAVEC;
        RETURN;

END STRTUP;

/************************ FNSHUP ************************/

        FNSHUP:  PROCEDURE PUBLIC;
/*_____________________________________________________

Make calls to RESTORC and COMPLETE for convenience.
_____________________________________________________*/

        CALL RESTORC;
        CMPLT;
        RETURN;

END FNSHUP;

/************************ ATTRIB ************************/

        ATTRIB:  PROCEDURE  PUBLIC;
/*_____________________________________________________

Put the Control Sequence for setting the cursor attribute
specified by the current contents of NUM into DSP$BUF.
_____________________________________________________*/

        INTRO;
        CALL PUT$NUM$1;
        STUFF('m');
        RETURN;

END ATTRIB;

/************************ POSITION ************************/
```

```
        POSITION:  PROCEDURE (ROW,COL) PUBLIC;
/*-------------------------------------------------------------

Control sequence for moving the cursor to the absolute
row and column position is entered into the display
buffer.
-------------------------------------------------------------*/

        DECLARE (ROW,COL) BYTE;

        INTRO;
        NUM = B2D(ROW);
        CALL PUT$NUM$2;
        DLMT;
        NUM$3 = B2D(COL);
        CALL PUT$NUM$3;
        STUFF('H');
        RETURN;

END POSITION;

/************************ JUMPUP ************************/

        JUMPUP:  PROCEDURE (PNTR) PUBLIC;
/*-------------------------------------------------------------

The code for the following is placed in the display
buffer:

        Position to the coordinates given in the
        Coordinate structure (COOR$STRC) pointed to by
        PNTR.

        Set the cursor to the attribute specified in
        COOR$STRC.ATTR.

NOTE:  In this package "JUMP" = "CALL JUMPUP".
-------------------------------------------------------------*/

        DECLARE PNTR ADDRESS;
        DECLARE COOR BASED PNTR COOR$STR;

        CALL POSITION(COOR.ROW, COOR.COL);
        NUM = COOR.ATTR;
        CALL ATTRIB;
        RETURN;

END JUMPUP;

/************************ HOR$LINE ************************/
```

```
        HOR$LINE:  PROCEDURE (LEN) PUBLIC;
/*-------------------------------------------------------------

Codes for drawing a horizontal line of length LEN at the
current position are entered into the display buffer.

NOTE:  In this package "HLINE" = "CALL HOR$LINE".
-------------------------------------------------------------*/

        DECLARE (I, LEN) BYTE;

        DO I = 1 TO LEN;
                STUFF(HL);
            END;
        RETURN;

END HOR$LINE;

/************************ DLDIAG ************************/

        DLDIAG: PROCEDURE  PUBLIC;
/*-------------------------------------------------------------

Down and Left Diagonal Move codes placed in DSP$BUF by
this routine.

NOTE:  In this package "LFBS" = "CALL DLDIAG".
       LFBS stands for Line Feed and Back Space.
-------------------------------------------------------------*/

        STUFF(LF);
        STUFF(BS);
        RETURN;

END DLDIAG;

/************************ VER$LINE ************************/

        VER$LINE:  PROCEDURE (HGT) PUBLIC;
/*-------------------------------------------------------------

Same as HOR$LINE except for vertical line of height HGT.

NOTE:  In this package "VLINE" = "CALL VER$LINE".
-------------------------------------------------------------*/

        DECLARE (I, HGT) BYTE;

        DO I = 1 TO HGT;
```

```
                LFBS;
                STUFF(VL);
            END;
        RETURN;

END VER$LINE;

/************************ CORNER ************************/

        CORNER:  PROCEDURE (TYPE) PUBLIC;
/*_____________________________________________________________

Place the code for the specified type graphics corner or
line intersection in the display buffer.

NOTE:  In this package "CRNR" = "CALL CORNER".
_____________________________________________________________*/

        DECLARE TYPE BYTE;
        DECLARE BAS$CRNR LITERALLY '152Q';
        DECLARE BAS$TEE LITERALLY '164Q';

        IF TYPE < 5 THEN
                STUFF(BAS$CRNR + TYPE);
        ELSE
                STUFF(BAS$TEE - 5 + TYPE);
        RETURN;

END CORNER;

/************************ SELECT ************************/

        SELECT:  PROCEDURE (CHAR$SET) PUBLIC;
/*_____________________________________________________________

Put the seqeunce for designating the character set into
the display buffer.

        CHAR$SET        Character Set Selected

           0                    UK
           1                    ASCII
           2                    Graphics
           3                    Alternate
           4                    Special Graphics
_____________________________________________________________*/

        DECLARE CHAR$SET BYTE;
        DECLARE CS$SEL (*) BYTE DATA('AB012');
```

```
        ESCP;
        STUFF('(');
        STUFF(CS$SEL(CHAR$SET));
        RETURN;

END SELECT;

/************************** BOX **************************/

        BOX:  PROCEDURE (PNTR) PUBLIC;
/*_____________________________________________________________

Draw a box as specified in the BOX$SPEC$STR pointed to
by PNTR.  See the description of BOX$SPEC$STR at the start
of this package.  This routine will actually send the box
to the console.  The cursor is restored to its initial
position and attributes.
                                                            */
_____________________________________________________________

        DECLARE PNTR ADDRESS;
        DECLARE BOX$S BASED PNTR BOX$SPEC$STR;

        CALL STRTUP;
        JUMP(PNTR);
        CALL SELECT(GRAPH);
        STUFF(' ');
        HLINE(BOX$S.LEN);
        CRNR(UR);
        VLINE(BOX$S.HGT);
        JUMP (PNTR);
        CRNR(UL);
        VLINE(BOX$S.HGT);
        LFBS;
        CRNR(LL);
        HLINE(BOX$S.LEN);
        CRNR(LR);
        CALL FNSHUP;
        RETURN;

END BOX;

/************************** CRLF **************************/

        CRLF:   PROCEDURE;

        CALL DISPLAY(.(CR),1);
        CALL DISPLAY(.(LF),1);
        RETURN;

END CRLF;
```

```
/************************ STRING ************************/

        STRING: PROCEDURE (PNTR,COUNT) PUBLIC;
/*_____________________________________________________________

Transfer the ASCII string pointed to by PNTR and of length
given by COUNT into the display buffer.
_____________________________________________________________*/

        DECLARE (COUNT,I) BYTE;
        DECLARE PNTR ADDRESS;
        DECLARE DAT$BUF BASED PNTR (1) BYTE;

        DO I = 0 TO (COUNT-1);
                STUFF(DAT$BUF(I));
           END;
        RETURN;

END STRING;

/************************ DSP$LBL ************************/

        DSP$LBL: PROCEDURE (PNTR) PUBLIC;
/*_____________________________________________________________

Place the position control sequence and ASCII data needed
to display a character string at a specific screen location
in the display buffer.  The position coordinates and the
length and pointer for the ASCII string are given by the
contents of the LABEL$STR pointed to by PNTR.  See the
description of LABEL$STR at the beginning of this package.
_____________________________________________________________*/

        DECLARE PNTR ADDRESS;
        DECLARE LBL BASED PNTR LABEL$STR;

        CALL POSITION(LBL.ROW,LBL.COL);
        CALL STRING(LBL.PNTR,LBL.COUNT);
        RETURN;

END DSP$LBL;

/************************ NUMBER ************************/

        NUMBER: PROCEDURE (NMB) PUBLIC;
/*_____________________________________________________________

First convert the 16 bit hex value in NMB to decimal then
convert the resulting four decimal digits to ASCII.  Blank
```

```
the leading zeros in the ASCII representation and transfer
the result into the display buffer.
                                                              */

        DECLARE NMB ADDRESS;
        DECLARE DAT$BUF (4) BYTE;

        CALL ASCII$4(B2D(NMB),.DAT$BUF);
        CALL BLANK(.DAT$BUF,3);
        CALL STRING(.DAT$BUF,4);
        RETURN;

END NUMBER;

/*********************** PLACE$NMB ***********************/

        PLACE$NMB: PROCEDURE (PNTR,NMB) PUBLIC;
/*
Place a position control sequence followed by the ASCII
representation of the value of NMB in the display buffer.
The position control sequence will effectively place the
value of NMB at the screen coordinates given by the
DATA$COOR$STR pointed to by PNTR.  See description of
this structure earlier.
                                                              */

        DECLARE (PNTR,NMB) ADDRESS;
        DECLARE NUM$COOR BASED PNTR DATA$COOR$STR;

        CALL POSITION(NUM$COOR.ROW,NUM$COOR.COL);
        CALL NUMBER(NMB);
        RETURN;

END PLACE$NMB;

/************************ DSP$NUM ************************/

        DSP$NUM: PROCEDURE (PNTR,NMB) PUBLIC;
/*
Display the decimal representation of the hex value stored
in NMB at the screen coordinates given by the DATA$COOR$STR
pointed to by PNTR.  This routine actual outputs the value
to the console.  The initial cursor position and attributes
are restored.
                                                              */

        DECLARE (PNTR,NMB) ADDRESS;
```

```
        CALL STRTUP;
        CALL PLACE$NMB(PNTR,NMB);
        CALL FNSHUP;
        RETURN;

END DSP$NUM;

/********************** DSP$2$NUMS **********************/

        DSP$2$NUMS: PROCEDURE (COOR$PNTR,VAL$PNTR) PUBLIC;
/*_____________________________________________________________

Display the 2 values in the 2 byte array pointed to by
VAL$PNTR at the respective coordinates specified by the
2 structure array pointed to by COOR$PNTR.
_____________________________________________________________*/

        DECLARE (COOR$PNTR,VAL$PNTR) ADDRESS;
        DECLARE COOR BASED COOR$PNTR (2) DATA$COOR$STR;
        DECLARE VAL  BASED VAL$PNTR  (2) BYTE;

        CALL STRTUP;
        IF RDR$MODE = CAL THEN
            DO;
                NUM = 4;
                CALL ATTRIB;
            END;
        CALL PLACE$NMB(.COOR(0),VAL(0));
        IF RDR$MODE = CAL THEN
                NUM = 0;
        ELSE
                NUM = 4;
        CALL ATTRIB;
        CALL PLACE$NMB(.COOR(1),VAL(1));
        CALL FNSHUP;
        RETURN;

END DSP$2$NUMS;

/********************** DSP$RANGE **********************/

        DSP$RANGE: PROCEDURE PUBLIC;
/*_____________________________________________________________

Display the range delay data at its proper coordinates.
_____________________________________________________________*/

        DECLARE RANGE$CHAR ADDRESS;

        IF SDL$MODE = LOCAL THEN
```

```
            CALL DSP$2$NUMS(.RD$COOR,.RANGE$VAL);
         ELSE
            DO;
               CALL DISPLAY(.('RD'),2);
               RANGE$CHAR = ASCII$2(RANGE$VAL(0));
               CALL DISPLAY (.RANGE$CHAR,2);
               RANGE$CHAR = ASCII$2(RANGE$VAL(1));
               CALL DISPLAY (.RANGE$CHAR,2);
               CALL CRLF;
            END;
         RETURN;

END DSP$RANGE;

/************************ DSP$GATE ************************/

         DSP$GATE: PROCEDURE PUBLIC;
/*_____________________________________________________________

Display the gate width data at its proper coordinates.
_____________________________________________________________*/

         DECLARE GATE$CHAR ADDRESS;

         IF SDL$MODE = LOCAL THEN
            CALL DSP$2$NUMS(.GP$COOR,.GATE$VAL);
         ELSE
            DO;
               CALL DISPLAY(.('GW'),2);
               GATE$CHAR = ASCII$2(GATE$VAL(0));
               CALL DISPLAY (.GATE$CHAR,2);
               GATE$CHAR = ASCII$2(GATE$VAL(1));
               CALL DISPLAY (.GATE$CHAR,2);
               CALL CRLF;
            END;
         RETURN;

END DSP$GATE;

/************************ DSP$MODE ************************/

         DSP$MODE: PROCEDURE PUBLIC;
/*_____________________________________________________________

Display information on current radar mode of operation at
its proper screen coordinates.
_____________________________________________________________*/

         DECLARE CAL$BUF (*) BYTE DATA('CALIBRATE');
         DECLARE OPR$BUF (*) BYTE DATA(' OPERATE ');
```

```
        DECLARE CONT$BUF (*) BYTE DATA(' CONTINUOUS');
        DECLARE SNGL$BUF (*) BYTE DATA('   SINGLE  ');

        IF SDL$MODE = LOCAL THEN
           DO;
              CALL STRTUP;
              CALL POSITION(CO$COOR.ROW,CO$COOR.COL);
              IF RDR$MODE = CAL THEN
                 CALL STRING(.CAL$BUF,LENGTH(CAL$BUF));
              ELSE
                 CALL STRING(.OPR$BUF,LENGTH(OPR$BUF));
              IF REPS = CONT THEN
                 CALL STRING(.CONT$BUF,LENGTH(CONT$BUF));
              ELSE
                CALL STRING(.SNGL$BUF,LENGTH(SNGL$BUF));
              CALL FNSHUP;
           END;
        ELSE
           DO;
              CALL DISPLAY(.('MO'),2);
              IF RDR$MODE = CAL THEN
                 CALL DISPLAY(.('C'),1);
              ELSE
                 CALL DISPLAY(.('O'),1);
              IF REPS = CONT THEN
                 CALL DISPLAY(.('C'),1);
              ELSE
                CALL DISPLAY(.('S'),1);
              CALL CRLF;
           END;
        RETURN;

END DSP$MODE;

/*********************** DSP$GAIN ************************/

        DSP$GAIN: PROCEDURE PUBLIC;
/*_____________________________________________________

Display the current gain value at its proper coordinates.
_____________________________________________________*/

        DECLARE GAIN$CHAR ADDRESS;
        DECLARE BSWING BYTE;

        IF SDL$MODE = LOCAL THEN
           CALL DSP$NUM(.GN$COOR,GAIN$VAL);
        ELSE
           DO;
              IF RDR$MODE = CAL THEN
```

```
            CALL DISPLAY(.('GC'),2);
         ELSE
            DO;
               CALL VALDAT(.BSWING);
               IF BSWING THEN
                  CALL DISPLAY(.('GO'),2);
               ELSE
                  CALL DISPLAY(.('GX'),2);
            END;
         GAIN$CHAR = ASCII$2(GAIN$VAL);
         CALL DISPLAY (.GAIN$CHAR,2);
         CALL CRLF;
      END;
   RETURN;

END DSP$GAIN;

/************************ DSP$HEAD ************************/

   DSP$HEAD: PROCEDURE PUBLIC;
/*_____________________________________________________

Display the head and band information at the proper screen
coordinates.
_____________________________________________________*/

   DECLARE HEAD$VAL BYTE;

   IF SDL$MODE = LOCAL THEN
      DO;
         CALL STRTUP;
         CALL POSITION(HED$COOR.ROW,HED$COOR.COL);
         STUFF(ASCLO(HEAD));
         CALL POSITION(BND$COOR.ROW,BND$COOR.COL);
         STUFF(BAND);
         CALL FNSHUP;
      END;
   ELSE
      DO;
         HEAD$VAL = ASCLO(HEAD);
         CALL DISPLAY(.('HE'),2);
         CALL DISPLAY(.HEAD$VAL,1);
         CALL CRLF;
      END;
   RETURN;

END DSP$HEAD;

/************************ DSP$POL ************************/
```

```
        DSP$POL: PROCEDURE (CRCT) PUBLIC;
/*_____________________________________________________________

Display the current polarization at its proper coordinates
for the specified circuit (Tx or Rx).
_____________________________________________________________*/

        DECLARE CRCT BYTE;

        IF SDL$MODE = LOCAL THEN
           DO;
              CALL STRTUP;
              IF CRCT = TXC THEN
                 DO;
                    CALL POSITION(TXP$COOR.ROW,TXP$COOR.COL);
                    STUFF(TXP$CHAR);
                 END;
              ELSE
                DO;
                   CALL POSITION(RXP$COOR.ROW,RXP$COOR.COL);
                   STUFF(RXP$CHAR);
                END;
              CALL FNSHUP;
           END;
        ELSE
           DO;
              IF CRCT = TXC THEN
                 DO;
                    CALL DISPLAY(.('TP'),2);
                    CALL DISPLAY(.TXP$CHAR,1);
                 END;
              ELSE
                 DO;
                    CALL DISPLAY(.('RP'),2);
                    CALL DISPLAY(.RXP$CHAR,1);
                 END;
              CALL CRLF;
           END;
        RETURN;

END DSP$POL;

/************************ CMD$ERR ************************/

        CMD$ERR: PROCEDURE PUBLIC;
/*_____________________________________________________________

Display the "Command Tail Error" message on the console.
_____________________________________________________________*/
```

```
        CALL DISPLAY
                (.('%Command Tail Error.',CR,LF,'%'),0);
        RETURN;

END CMD$ERR;

/************************ INICRT ************************/

        INI$CRT: PROCEDURE  PUBLIC;
/*_____________________________________________________

Initialize the consol display.
_____________________________________________________*/

        DECLARE K BYTE;

        CALL DISPLAY(.('%',ESC,'[2J',ESC,'[10;24r',
                        ESC,'[?3h%'),0);
        CALL RESTRT;
        JUMP (.SEP$LINE);
        CALL SELECT(GRAPH);
        HLINE (115);
        CALL SELECT(SPEC);
        JUMP (.LOGO$COOR);
        CMPLT;
        CALL DISPLAY(.LOGO,LENGTH(LOGO));
        DO K = 0 TO LAST(BOX$SPEC);
                CALL BOX(.BOX$SPEC(K));
            END;
        CALL RESTRT;
        CALL SELECT(NORM);
        DO K = 0 TO 3;
                CALL DSP$LBL(.LINE$LBL(K));
            END;
        CMPLT;
        CALL RESTRT;
        DO K = 4 TO LAST(LINE$LBL);
                CALL DSP$LBL(.LINE$LBL(K));
            END;
        JUMP (.SCROLL$REG);
        CMPLT;
        CALL DISPLAY(.('%REMOTE OPERATIONS UTILITY',
                        '  -  RMX 80 CLI V1.0',CR,LF,
                        '---> Local Mode',CR,LF,'%'),0);

        RETURN;

END INI$CRT;

/************************************************************/
```

```
END RODFAT;      /* End of module. */
EOF
```

```
/*****************************************************************/
/*****************************************************************/
/*                                                               */
/*                        S Y S U T L                            */
/*                                                               */
/*  System Utilities Library          Developed by C. Kronke     */
/*                                                               */
/*****************************************************************/
/*****************************************************************/


ASCII2: DO;
/*_______________________________________________________________*/
COMB:           PROCEDURE (HI$BYTE,LO$BYTE) ADDRESS EXTERNAL;
                DECLARE (HI$BYTE,LO$BYTE) BYTE;
        END COMB;
ASCLO:          PROCEDURE (DIGIT) BYTE EXTERNAL;
                DECLARE DIGIT BYTE;
        END ASCLO;
ASCHI:          PROCEDURE (DIGIT) BYTE EXTERNAL;
                DECLARE DIGIT BYTE;
        END ASCHI;
/*_______________________________________________________________*/
        ASCII$2:  PROCEDURE (DIGS$VAL) ADDRESS PUBLIC;
/*---------------------------------------------------------------
        Converts a two digit input value into two ASCII
        code bytes and returns the combined results.

        Form of call:
                        ascii$words =  ASCII$2(in$val)
        where,
                in$val is a byte variable containing the two
                digit value to be converted,
        and
                ascii$word is an address variable that will
                be given the resulting two codes.

        The returns codes are in reverse order to the way they
        are input.   This is so that when they are actually
        stored in ascii$words they will be in the correct order.

---------------------------------------------------------------*/
                DECLARE DIGS$VAL BYTE;
/*
                -- Convert both digits to ASCII and combine
                        the results to get the return value.  --
*/
                RETURN COMB(ASCLO(DIGS$VAL),ASCHI(DIGS$VAL));
/* */
```

```
        END  ASCII$2;
/*                                                                    */
END ASCII2;     /* End of module. */
EOF



ASCII4: DO;
/*--------------------------------------------------------------------*/
ASCII$2:        PROCEDURE (DIGS$VAL) ADDRESS EXTERNAL;
                DECLARE DIGS$VAL BYTE;
        END ASCII$2;
/*                                                                    */
        ASCII$4:  PROCEDURE (WORD,BUF$ADDR) PUBLIC;
/*--------------------------------------------------------------------
        This procedure will convert a four digit value to
        the four ASCII codes that represent the digits.

        Form of call:
                        CALL ASCII$4(word,buf$addr)
        where,
                word is an address variable containing the four
                digit value to be converted to ASCII,
        and
                buf$addr is an address variable that points to
                the beggining of a four byte buffer where the
                converted data is to be stored.


--------------------------------------------------------------------*/
                DECLARE (WORD, BUF$ADDR) ADDRESS;
                DECLARE BUFFER BASED BUF$ADDR (2) ADDRESS;
/*
                --  Convert the high byte to two ASCII codes
                        and store them first.   --
*/
                BUFFER(0) = ASCII$2(HIGH(WORD));
/*
                --  Now convert the low byte to two ASCII codes
                        and store them in next two bytes.  --
*/
                BUFFER(1) = ASCII$2(LOW(WORD));
/* */
                RETURN;
/* */
        END ASCII$4;
/*                                                                    */
END ASCII4;     /* End of module. */
EOF
```

```
ASCHI: DO;
/*_____________________________________________________________*/
        ASCLO: PROCEDURE (DIGIT) BYTE EXTERNAL;
                DECLARE DIGIT BYTE;
        END ASCLO;
/*_____________________________________________________________*/
/*_____________________________________________________________*/
        ASCHI: PROCEDURE (DIGIT) BYTE PUBLIC;
/*-------------------------------------------------------------
        This procedure performs the same operation as
        ASCLO except the high nibble of the input digit
        is converted to ASCII instead of the low nibble.

        Form of call:

                        ascii$code = ASCHI(digit$val)
        where,
                ascii$code and digit$val are the same as in
                ASCLO above except that the digit to be
                converted is taken from the upper four bits
                of digit$val.

-------------------------------------------------------------*/
                DECLARE DIGIT BYTE;
/*
                --  Put digit to be converted into
                        the lower bit positions and
                        use ASCLO to get ASCII code.  --
*/
                RETURN ASCLO(SHR(DIGIT,4));
/* */
        END ASCHI;
/*_____________________________________________________________*/
END ASCHI;      /* End of Module */
EOF


DISPLA: DO;
$INCLUDE (:F2:RMXGEN.INC)
$INCLUDE (:F2:THDINO.EXT)
/*-------------------------------------------------------------*/
        DECLARE CWR$RQ$MSG TH$REQ$MSG PUBLIC;
        DECLARE WRIT$RESP$X EXCHANGE$DESCRIPTOR PUBLIC;
/*-------------------------------------------------------------*/
        INITDSP:  PROCEDURE PUBLIC;
/*-------------------------------------------------------------
        INITDSP is a routine that must be called prior to the
        use of DISPLAY (next procedure).  This procedure will
        set up the message header needed for DISPLAY, create
```

```
        a response exchange for that message, and send a dummy
        response message to the write response exchange so that
        DISPLAY will not have to wait the first time.

        Form of call:
                        CALL INITDSP;
-------------------------------------------------------------*/
                CWR$RQ$MSG.LENGTH = 17;
                CWR$RQ$MSG.TYPE = WRITE$TYPE;
                CWR$RQ$MSG.RESPONSE$EXCHANGE = .WRIT$RESP$X;
                CWR$RQ$MSG.STATUS = 0;
                CALL RQCXCH(.WRIT$RESP$X);
/*@
                CALL RQSEND(.WRIT$RESP$X,.CWR$RQ$MSG);
@*/
                RETURN;

        END INITDSP;

/*-------------------------------------------------------------*/
        DISPLAY:  PROCEDURE (BUF$ADDR,COUNT) PUBLIC;
/*-------------------------------------------------------------
        DISPLAY is a utility that will be used to output
        an ASCII buffer to the system console.

        Form of call:
                        CALL DISPLAY(buf$addr,count)
        where,
                buf$addr is an address that points to the buffer
                to be output,
        and
                count is a byte variable that acts as either a
                control character or buffer length count.

        If the value of count is zero the first character in
        the buffer is taken as a delimeter and the input buffer
        is output untill the delimiter is encountered a second
        time.  Note that the delimiters are not output.  If the
        value of count is non-zero then the value is equivalent
        to the number of characters that will be output.


-------------------------------------------------------------*/

        DECLARE COUNT    BYTE;
        DECLARE BUF$ADDR ADDRESS;
        DECLARE BUFFER BASED BUF$ADDR (1) BYTE;
        DECLARE SCRAP BYTE;

/*              --  Test count for zero.  --                  */
```

```
                IF COUNT = 0 THEN
/*              -- If count is zero find actual count by
                        by scanning buffer for delimiters.  --*/
                   DO;
                        COUNT = 1;
                        DO WHILE BUFFER(COUNT) <> BUFFER(0);
                                COUNT = COUNT+1;
                           END;
                        COUNT = COUNT-1;
                        BUF$ADDR = BUF$ADDR+1;
                     END;
/*              --  Wait for the Request Message to be
                        available for use.              --*/
/*@
                SCRAP = RQWAIT(.WRIT$RESP$X,0);
@*/
/*              -- Set-up output request message data.  --*/
                CWR$RQ$MSG.BUFFER$ADDR = BUF$ADDR;
                CWR$RQ$MSG.COUNT = COUNT;
/*              -- Now send the request to the Terminal
                        Handler output exchange.        --*/
                CALL RQSEND(.RQOUTX,.CWR$RQ$MSG);
/*              --  Wait for the Request Message to be
                        available for use.              --*/
                SCRAP = RQWAIT(.WRIT$RESP$X,0);
                RETURN;
/* */
        END DISPLAY;
/*--------------------------------------------------------------*/
END DISPLA;     /* End of module. */
EOF


PACK: DO;
/*______________________________________________________________*/
        PACK:   PROCEDURE (HI$NIB,LO$NIB) BYTE PUBLIC;
/*--------------------------------------------------------------
        This procedure will pack two nibbles together into
        one byte.

        Form of call:
                        byte$value = PACK(hi$nib,lo$nib)
        where,
                hi$nib and lo$nib are byte variables containing
                the nibbles to go into the high and low nibbles
                of the result respectively,
        and
                byte$value is an address variable to which the
                result is to be returned.
```

```
        The nibbles are assumed to be in the low half of both
        input variables.  The high half of the inputs is mask
        off and thus ignored.

-----------------------------------------------------------*/
                DECLARE (HI$NIB,LO$NIB) BYTE;
/*
                --  Simply shift hi$nib to proper bit
                    coordinates,  mask the nibbles,
                    and return them combined.
*/
                RETURN (SHL(HI$NIB,4) OR (LO$NIB AND 0FH));
/* */
        END  PACK;
/*_____________________________________________________________*/
END PACK;       /* End of module. */
EOF



        NAME    COMB
;
        CSEG
;
;===============================================================
;
;       Function COMB  -  Combine Bytes Routine
;
; Inputs:       C - High byte of data.
;               E - Low byte of data.
; Outputs:      HL - Combined word.
; Calls:        Nothing.
; Destroys:     HL.
; Description:  This routine will provide a simple PL/M function
;               that combines two byte parameters into a single
;               address parameter.
;
; PL/M Call Syntax:
;
;               addr$var = COMB(hi$byte,lo$byte)
;       where:
;               hi$byte and lo$byte are the byte values to be
;               combined,
;       and
;               addr$var is an address variable in which the
;               resulting address value is to be returned.
;_______________________________________________________________
;
        PUBLIC          COMB
```

```
;_____________________________________________________________;
COMB:
        MOV     L,E     ; E to low byte of HL
        MOV     H,C     ;  and C to high byte of HL.
        RET             ; Its that simple.
;
;_____________________________________________________________
;
        END
B2D: DO;
/*                                                             */
COMB:           PROCEDURE (HI$BYTE,LO$BYTE) ADDRESS EXTERNAL;
                DECLARE (HI$BYTE,LO$BYTE) BYTE;
        END COMB;
/*_____________________________________________________________*/
        B2D: PROCEDURE (BIN) ADDRESS PUBLIC;
/*-------------------------------------------------------------
        This procedure converts a binary integer (BIN) to a
        binary coded decimal integer.  The input value is
        expected to be a binary integer between 0 H and
        270F H.  The output value will therefore be a
        binary coded decimal integer in the range of 0 to
        9,999.

        Form of call:
                        decnum = B2D(binnum)
        where,
                binnum is an address variable with the
                binary value to be converted
        and
                decnum is an address variable to which
                the decimal value is to be returned.

        Note:  No check is made for the validity of the input.
-------------------------------------------------------------*/
                DECLARE BIN ADDRESS, (LODIGS,HIDIGS) BYTE;
/*
                -- Calculate the thousands decimal digit. --
*/
                        HIDIGS = SHL(LOW(BIN/1000),4);
                        BIN = BIN MOD 1000;  /* The remainder */
/*
                -- Next find the hundreds decimal digit and
                   combine it with the thousands digit. --
*/
                        HIDIGS = HIDIGS OR LOW(BIN/100);
                        BIN = BIN MOD 100;   /* The remainder */
/*
                -- Calculate the tens and ones decimal
```

```
                    digits and combine them as above. --
*/
                        LODIGS = SHL(LOW(BIN/10),4)
                                        OR LOW(BIN MOD 10);
/*
                -- Combine the two high order digits with
                    the two low order digits to form the
                    return value. --
*/
                RETURN COMB(HIDIGS,LODIGS);
/* */
        END B2D;
/*_____________________________________________________________*/
END B2D;        /* End of module. */
EOF



HEX: DO;

/*_____________________________________________________________*/
        HEX:  PROCEDURE (DIGIT) BYTE PUBLIC;
/*-------------------------------------------------------------
        This procedure will convert an ASCII code to its
        HEX or BCD value.  A check is made for valid inputs.

        Form of call:
                        hex$digit = HEX(ascii$code)
        where,
                ascii$code  is a byte  value containing  the
                ASCII code of the  proposed hex digit.   The
                code  will be tested  to determine  if it is
                a valid  input.   If so then  the hex  digit
                will be returned in hex$digit.  If it is not
                then OFFH will be returned in hex$digit.

-------------------------------------------------------------*/
                DECLARE DIGIT BYTE;
/*
                --  If ASCII code is greater then '@' then
                 digit should be A-F so subtract bias of 7.  --
*/
                        IF DIGIT > 40H THEN
                                DIGIT = DIGIT - 7;
/*
                -- If not greater then '@' but is greater
                        then '9' then input is invalid.  --
*/
                        ELSE IF DIGIT > 3AH THEN
                                GOTO ERROR;
```

```
/*
                -- If adjusted code is inside valid range
                 subtract final bias and return the result. --
*/
                        IF DIGIT < 40H AND DIGIT > 2FH THEN
                                RETURN (DIGIT - 30H);
/*
                -- Else the input is invalid. --
*/
                ERROR:
                                RETURN 0FFH;
                END HEX;
END;
BLANK: DO;
/*_____________________________________________________________*/
        BLANK:  PROCEDURE (BUF$ADDR,COUNT) PUBLIC;
/*-------------------------------------------------------------
        BLANK scans an ASCII buffer for leading zeros and
        converts them to the ASCII code for a blank.

        Form of call:
                        CALL BLANK(buf$addr,count)
        where,
                buf$addr is an address variable that points to
                the buffer to be scanned, and count is the max.
                number of leading zeros to be blanked.

-------------------------------------------------------------*/
                DECLARE BUF$ADDR ADDRESS, (COUNT, I) BYTE;
                DECLARE BUFFER BASED BUF$ADDR (10) BYTE;
/*
        Note:  Although the above declare is for a buffer of
        only ten bytes the buffer to be scanned is not limit
        to ten since BUFFER is a based variable.

                -- Iterate untill non-zero code encountered
                        or untill the count expires. --
*/
                I = 0;
                DO WHILE (BUFFER(I) = '0') AND (I < COUNT);
                        BUFFER(I) = ' ';
                        I = I+1;
                END;
                RETURN;
/* */
        END BLANK;
/*_____________________________________________________________*/
END BLANK;      /* End of module. */
EOF
```

```
ASCLO: DO;
/*_____________________________________________________________*/
        ASCLO: PROCEDURE (DIGIT) BYTE PUBLIC;
/*-------------------------------------------------------------
        This procedure converts a single Hexadecimal or
        Binary Coded Decimal digit stored in the lower
        nibble of a byte variable to its ASCII code.

        Form of call:
                        ascii$code = ASCLO(digit$val)
        where,
                digit$val is a byte variable that contains
                the actual digit to be converted in its lower
                four bits,
        and
                ascii$code is a byte variable in which the
                resulting ASCII code is to stored.

        The upper four bits are automatically mask off.

-------------------------------------------------------------*/
                DECLARE DIGIT BYTE;
/*
                -- Mask off upper nibble and add
                        ASCII base offset of 30H.  --
*/
                        DIGIT = (DIGIT AND 0FH) + 30H;
/*
                -- Now adjust result if input was
                        Hex digit greater than 9.
                        Return the proper value.   --
*/
                        IF DIGIT > 39H THEN
                                RETURN (DIGIT + 7);
                        ELSE
                                RETURN DIGIT;
/* */
        END ASCLO;
/*_____________________________________________________________*/
END ASCLO;      /* End of module. */
EOF


CROUT: DO;
/*-------------------------------------------------------------*/
DISPLAY:  PROCEDURE (BUF$ADDR,COUNT) EXTERNAL;
                DECLARE COUNT    BYTE;
                DECLARE BUF$ADDR ADDRESS;
        END DISPLAY;
```

```
/*-------------------------------------------------------------*/
        CROUT:  PROCEDURE PUBLIC;
/*-------------------------------------------------------------
        Output Carraige return and line feed to terminal.
-------------------------------------------------------------*/
        CALL DISPLAY(.(0DH,0AH),2);
        RETURN;
/* */
        END CROUT;
/*-------------------------------------------------------------*/
END CROUT;      /* End of module. */
EOF


D2B: DO;

/*_____________________________________________________________*/
        D2B: PROCEDURE (BCD) ADDRESS PUBLIC;
/*-------------------------------------------------------------
        This procedure converts a binary coded decimal
        integer (BCD) to a binary integer  The input
        value is expected to be a decimal integer between
        0 and 9,999.  The output value will be the
        corresponding binary integer for that number
        and would therefore range from 0 H to 270F H.

        Form of call:
                        binnum = D2B(decnum)
        where,
                decnum is an address variable with the
                decimal value to be converted
        and
                binnum is an address variable to which
                the binary value is to be returned.

        Note:  No check is made for the validity of the input.

-------------------------------------------------------------*/
                DECLARE (BCD,BIN) ADDRESS, DIGIT BYTE;
                DECLARE THOUS (10) ADDRESS DATA
                        (0,1000,2000,3000,4000,5000,6000,7000,
                         8000,9000);
                DECLARE HUNDS (10) ADDRESS DATA
                        (0,100,200,300,400,500,600,700,800,900);
                DECLARE TENS (10) BYTE DATA
                        (0,10,20,30,40,50,60,70,80,90);
/*
                --  Find the binary value for the
                    thousands digit and save in BIN.  --
```

```
*/
                        DIGIT = SHR(HIGH(BCD),4);
                         BIN = THOUS(DIGIT);
/*
                -- Find the binary value of the hundreds
                   digit and add it to BIN.  --
*/
                        DIGIT = HIGH(BCD) AND OFH;
                         BIN = BIN + HUNDS(DIGIT);
/*
                -- Find the binary value of the tens and
                   ones digits and add these values to
                   BIN to get the returned binary value.  --
*/
                        DIGIT = SHR(LOW(BCD),4);
/* */
                RETURN (BIN + TENS(DIGIT) + (LOW(BCD) AND OFH));
/* */
        END D2B;
END;  /* end module  */
EOF
```

```
        NAME    HEDCON
; HEDCON:       Head Control Software
;
; ************************************************************
; *                                                          *
; ************************************************************
; *                                                          *
; *     The following routines provide the required code     *
; *     for controlling up to four radar heads on the        *
; *     Land Based Radar Polarimeter System.  Utilities      *
; *     provided include software for controlling and        *
; *     monitoring transfer switches in the radar heads.     *
; *     These switches are intended for use in selecting     *
; *     transmit and receive polarizations and for           *
; *     setting the heads in calibrate or operate modes.     *
; *                                                          *
; ************************************************************
; *                                                          *
; ************************************************************
;
;
;
;               Port Address Equates
;               --------------------
CALCON  EQU     0E8H    ; Calibration Control - Port A of
                        ; the 8255 programmed for Mode 0
                        ; output operation.
;
INDENB  EQU     0E9H    ; Indicator Enable - Port B of
                        ; the 8255 programmed for Mode 0
                        ; output operation.
;
RFHCON  EQU     0EBH    ; RF Head General Control - Port
                        ; C (lower half only) of the 8255
                        ; programmed for Mode 0 output.
                        ; [NOTE: 0EBH is not the address
                        ; of Port C!  It is intended to
                        ; use the Bit Set/Reset feature
                        ; of the 8255 and 0EBH is there-
                        ; fore the address of the 8255's
                        ; control port.)
;
INDRET  EQU     0EAH    ; Indicator Return - Port C (upper
                        ; half only) of the 8255 set for
                        ; Mode 0 input operation.
;
PPICON  EQU     0EBH    ; Programmable Peripheral Interface
                        ; Control Register - Command port
                        ; of the 8255 device.
```

```
;
;-------------------------------------------------------------
;
;                Constant Equates
;                ----------------
PPICMD  EQU  10001000B  ; PPI Command Word used to program
                        ; 8255 as follows:
                        ;       GROUP A - Mode 0
                        ;           Port A - Output
                        ;           Port C Upper - Input
                        ;       GROUP B - Mode 0
                        ;           Port B - Output
                        ;           Port C Lower - Output
;
HORRXP  EQU  00000110B  ; Horizontal Receive Polarization
                        ; control word used to reset the
                        ; Receive Polarization Control bit
                        ; (Port EA, Bit 3, J2-18) to off,
                        ; selecting horizontal polarization.
;
VERRXP  EQU  00000111B  ; Vertical Receive Polarization
                        ; control word used to set the
                        ; Rx. Polarization Control bit to
                        ; on, selecting vertical pol.
;
HORTXP  EQU  00000100B  ; Horizontal Transmit Polarization
                        ; control word used to reset the
                        ; Tx. Pol. Control bit (Port EA,
                        ; Bit 2, J2-20) to off, selecting
                        ; horizontal pol.
;
VERTXP  EQU  00000101B  ; Vertical Transmit Polaization
                        ; control word used to set the
                        ; Tx. Pol. Control bit to on,
                        ; selecting vertical pol.
;
RF1OFF  EQU  00000010B  ; Control word used to turn the
                        ; Reserved RF Control 1 signal
                        ; bit (Port EA, Bit 1, J2-22) off.
;
RF1ON   EQU  00000011B  ; Control word used to turn the
                        ; Reserved RF Control 1 signal
                        ; bit on.
;
RF0OFF  EQU  00000000B  ; Control word used to turn the
                        ; Reserved RF Control 0 signal
                        ; bit (Port EA, Bit 0, J2-24) off.
;
RF0ON   EQU  00000001B  ; Control word used to turn the
                        ; Reserved RF Control 0 signal
```

```
                        ; bit on.
;
;=============================================================
;               UTILITIES
;-------------------------------------------------------------
;
;       Public Declerations
;
        PUBLIC  HEDCON, INIHED, RXPCON, TXPCON, RFOCON
        PUBLIC  RF1CON, GENPAT, ENBIND, GETPOL, GETCAL
        PUBLIC  COPCON
;
;-------------------------------------------------------------
;
;       Start of HEDCON Module
;
        CSEG
HEDCON:                 ; Label to identify start of module.
;
;-------------------------------------------------------------
;
;       INIHED  -  Initialize the PPI.
;
;       No inputs or outputs.  Must be called prior to
;       using any of the other routines that use the 8255.
;
INIHED:
        MVI   A,PPICMD  ; Simply load output register with
                        ;  command word defined above.
        OUT     PPICON  ; Output the command to the Control
                        ;  register of the 8255 PPI.
        MVI     A,0     ; Load output register with zero.
        OUT     INDENB  ; Output to Indicator Enable port to
                        ;  disable all indicator circuits.
        OUT     RFHCON  ; Output to RF Head General Control
                        ;  port to turn all those bits off.
        OUT     CALCON  ; Output to Calibration Control port
                        ;  to turn off all bits.
        RET             ;
;
;-------------------------------------------------------------
;
;       RXPCON  -  Receive Polarization Controller
;
;  Function:    Set the receiver polarization transfer
;                switch to desired polarization.
;  Inputs:      C - Desired Polarization: 0 = Horizontal,
;                                         Not 0 = Vertical.
;  Outputs:     None.
;  Destroys:    A and Flags.
```

```
;  Calls:       Nothing.
;
RXPCON:
        MOV     A,C     ; Transfer desired pol. to A.
        ANA     A       ; Test for A = 0 (Horizontal).
        JZ      RXPC10  ; If it is 0 then set polarization
                        ;  to horizontal.
RXPC05:                 ; Else set to vertical.
        MVI   A,VERRXP  ; Load output register with bit
                        ;  set control word for the Rx Pol.
                        ;  control bit.
        OUT     RFHCON  ; Output the control word to the
                        ;  8255 control port.
        RET             ; All done, return.
RXPC10:                 ;
        MVI   A,HORRXP  ; Load output register with bit
                        ;  reset control word.
        OUT     RFHCON  ; Output word to 8255 control port.
        RET             ; All done, Return.
;
;-----------------------------------------------------------
;
;       TXPCON  -  Transmit Polarization Controller
;
;  Function:    Set the transmitter polarization transfer
;                switch to desired polarization.
;  Inputs:      C - Desired Polarization: 0 = Horizontal,
;                                         Not 0 = Vertical.
;  Outputs:     None.
;  Destroys:    A and Flags.
;  Calls:       Nothing.
;
TXPCON:
        MOV     A,C     ; Transfer desired pol. to A.
        ANA     A       ; Test for A = 0 (Horizontal).
        JZ      TXPC10  ; If it is 0 then set polarization
                        ;  to horizontal.
TXPC05:                 ; Else set to vertical.
        MVI   A,VERTXP  ; Load output register with bit
                        ;  set control word for the Tx Pol.
                        ;  control bit.
        OUT     RFHCON  ; Output the control word to the
                        ;  8255 control port.
        RET             ; All done, return.
TXPC10:                 ;
        MVI   A,HORTXP  ; Load output register with bit
                        ;  reset control word.
        OUT     RFHCON  ; Output word to 8255 control port.
        RET             ; All done, Return.
;
```

```
;-------------------------------------------------------
;
;       RF0CON  -  RF Reserved Signal 0 Controller
;
;  Function:    Set the Reserved RF Control 0 signal to the
;                desired level (ON or OFF).
;  Inputs:      C - Desired level:  0 = OFF (low level),
;                                   Not 0 = ON (high level).
;  Outputs:     None.
;  Destroys:    A and Flags.
;  Calls:       Nothing.
;
RF0CON:
        MOV     A,C     ; Transfer desired level to A.
        ANA     A       ; Test for A = 0 (OFF).
        JZ      RF0C10  ; If it is zero set signal to off.
RF0C05:                 ; Else set signal to on.
        MVI   A,RF0ON   ; Load bit set control word.
        OUT     RFHCON  ; Output to 8255 control port.
        RET             ; All done, return.
RF0C10:                 ;
        MVI   A,RF0OFF  ; Load bit reset control word.
        OUT     RFHCON  ; Output to 8255 control port.
        RET             ; All done, return.
;
;-------------------------------------------------------
;
;       RF1CON  -  RF Reserved Signal 1 Controller
;
;  Function:    Set the Reserved RF Control 1 signal to the
;                desired level (ON or OFF).
;  Inputs:      C - Desired level:  0 = OFF (low level),
;                                   Not 0 = ON (high level).
;  Outputs:     None.
;  Destroys:    A and Flags.
;  Calls:       Nothing.
;
RF1CON:
        MOV     A,C     ; Transfer desired level to A.
        ANA     A       ; Test for A = 0 (OFF).
        JZ      RF1C10  ; If it is zero set signal to off.
RF1C05:                 ; Else set signal to on.
        MVI   A,RF1ON   ; Load bit set control word.
        OUT     RFHCON  ; Output to 8255 control port.
        RET             ; All done, return.
RF1C10:                 ;
        MVI   A,RF1OFF  ; Load bit reset control word.
        OUT     RFHCON  ; Output to 8255 control port.
        RET             ; All done, return.
;
```

```
;-------------------------------------------------------
;
;       GENPAT  -  Generate Circuit Control Bit Pattern
;
;  Function:    Used to translate a Head No. and Circuit
;                No. to a specific bit pattern which can be
;                used to turn on a corresponding circuit
;                (port bit) and turn all others (bits) off.
;  Inputs:      C - Desired Head No. (1, 2, 3, or 4).
;               E - Desired Circuit No. (0 or 1).
;  Outputs:     A - CCBP (Circuit Control Bit Pattern).
;  Destroys:    A, C and Flags.
;  Calls:       Nothing.
;
;  Description:
;               This subroutine will convert the Desired
;               Head No. and Circuit No. into a Shift
;               Count which will in turn be used to create
;               a Circuit Control Bit Pattern (CCBP).
;               The following table shows the CCBPs for
;               the range of inputs.
;
;          ---------------------------------------------
;           Head No.   Circ. No.   Shift Count    CCBP
;          ---------------------------------------------
;               1          0             0       00000001
;                          1             1       00000010
;               2          0             2       00000100
;                          1             3       00001000
;               3          0             4       00010000
;                          1             5       00100000
;               4          0             6       01000000
;                          1             7       10000000
;          ---------------------------------------------
;
GENPAT:
        MOV     A,C     ; Transfer desired head no. to A.
        DCR     A       ; Subtract 1 to change 1-4 to 0-3.
        ANI  00000011B  ; Mask off to reject 4 thru F Hex.
        ADD     A       ; Double number to translate 0, 1,
                        ;  2, & 3 to 0, 2, 4, & 6.
        MOV     C,A     ; Save result in C fo a moment.
        MOV     A,E     ; Transfer Desired Circuit No. to A.
        ANI  00000001B  ; Mask off to reject 2 thru F Hex.
        ADD     C       ; Add this result to previous result
        MOV     C,A     ;  to get Shift Count and save in C.
        MVI A,00000001B ; Load register with basic pattern.
GENP10:                 ;
        RZ              ; If Shift Count is zero return this
                        ;  pattern.
```

```
        RLC             ; Rotate Pattern to left one bit.
        DCR     C       ; Decrement Shift Count.
        JMP     GENP10  ; Repeat till done.
;
;-------------------------------------------------------
;
;       ENBIND  -  Enable Indicator Circuit
;
;  Function:    Enable a specified indicator circuit for a
;                specified radar head.
;  Inputs:      C - Desired Head No. (1, 2, 3, or 4).
;               E - Desired Circuit No.:  0 = Tx. Pol./Cal.
;                                         1 = Rx. Pol.
;  Outputs:     None.
;  Destroys:    A, C, and Flags.
;  Calls:       GENPAT - Generate Pattern.
;
;  Description:
;               The routine first calls GENPAT to aquire a
;               pattern used to enable the desired indicator
;               circuit.  This pattern is output to INDENB,
;               the Indicator Enable port, to effectively
;               turn on the desired indicator circuit and
;               disable all others.  A low level voltage,
;               0 volts, is required to enable an indicator
;               circuit and a high level, 5 volts, would
;               disable the circuit.  This requires that
;               the output buffers used on the INDENB port
;               be inverting type buffers (see GENPAT for
;               patterns generated).
;
ENBIND:
        CALL    GENPAT  ; Get pattern for setting up
                        ;  indicator enable circuit.
        OUT     INDENB  ; Output the pattern to the
                        ;  Indicator Enable port.
        NOP             ; Delay
        NOP             ;       to allow
        NOP             ;                circuit to
        NOP             ;                           settle.
        RET             ; All done, return.
;
;-------------------------------------------------------
;
;       GETPOL  -  Get Polarization
;
;  Function:    Read the polarization indicator return
;                for a specified circuit and specified
;                radar head.
;  Inputs:      C  -  Desired Head No. (1, 2, 3, or 4).
```

```
;               E  -  Desired Circuit No.:  0 = Tx. Pol.
;                                           1 = Rx. Pol.
;  Outputs:     A  -  Indication Value:  0 = Horizontal,
;                                        1 = Vertical,
;                                        2 = Failure 1,
;                                        3 = Failure 2.
;  Destroys:    A, C, E, and Flags.
;  Calls:       ENBIND - Enable Indicator.
;
;  Description:
;       This routine will first enable the specified indica-
;       tor circuit for the specified radar head by calling
;       ENBIND with the required parameters.  Next it will
;       read the Indicator Return port, INDRET, and test the
;       return signal to determine the state of the desired
;       transfer switch.  A low voltage on an indicator
;       return signal would indicate a completed circuit.
;       A completed circuit on Indicator 0 should appear for
;       a switch in the horizontal state.  Like wise on
;       Indicator 1 for a vertical state.  If both Indicator
;       0 and 1 indicate the same state (closed or open) the
;       indicator system has failed.  Failure 1 indicates
;       both indicators are open circuits and Failure 2
;       indicates both are closed circuits.
;
GETPOL:
        CALL    ENBIND  ; Enable the proper indicator
                        ;  for the desired head.
        IN      INDRET  ; Read the indicator return signal.
        ANI  00110000B  ; Strip out the Pol. Ind. bits and
        RRC             ;  put them
        RRC             ;          in 1st &
        RRC             ;                2nd bit
        RRC             ;                      positions.
        DCR     A       ; Convert:  00B to 3,  01B to 0,
        ANI  00000011B  ;  10B to 1  and  11B to 2.
        RET             ; Return the value in A as the
                        ;  indication value.
;
;-----------------------------------------------------------
;
;       GETCAL  -  Get Calibrate/Operate State
;
;  Function:    Read the Cal. Xfer Indicator circuit for
;                a specified radar head to determine the
;                state of that Cal./Op. Transfer switch.
;  Inputs:      C  -  Desired Head No. (1, 2, 3, or 4).
;  Outputs:     A  -  Indication Value:  0 = Calibrate,
;                                        1 = Operate,
;                                        2 = Failure 1,
```

```
;                                          3 = Failure 2.
;  Destroys:    A, C, E, and Flags.
;  Calls:       ENBIND - Enable Indicator.
;
;  Description:
;       Operation of GETCAL is much the same as GETPOL
;       except that it operates specifically for the
;       Cal. Xfer Indicator circuit.  A closed circuit
;       on Indicator 0 should indicate a switch is in
;       the calibrate position and a closed ciruit on
;       Indicator 1 should indicate a switch is in the
;       operate position.  Failure indications are the
;       same as for GETPOL.
;
GETCAL:
        MVI     E,0       ; Set Desired Circuit No. to 0
                          ;  for Cal. Xfer Indicator.
        CALL    ENBIND    ; Enable Cal. Xfer Ind. circuit.
        IN      INDRET    ; Read the indicator return signal.
        ANI  11000000B    ; Strip out the Cal. Xfer Ind. bits
        RLC               ;  & put them in 1st
        RLC               ;               & 2nd bit positions.
        DCR     A         ; Convert:  00B to 3,  01B to 0,
        ANI  00000011B    ;  10B to 1  and  11B to 2.
        RET               ; Return the value in A as the
                          ;  indication value.
;
;-------------------------------------------------------------
;
;       COPCON  -  Calibrate/Operate Controller
;
;  Function:    Operate the Cal./Op. Transfer switch for
;                a specified radar head into a specified
;                position.
;  Inputs:      C - Desired Head No. (1, 2, 3, or 4).
;               E - Desired State:  0 = Calibrate mode,
;                                   1 = Operate mode.
;  Outputs:     None.
;  Destroys:    A, C, and Flags.
;  Calls:       GENPAT.
;
;  Description:
;       First the routine generates the require bit pattern
;       needed to set the Cal. Xfer Control port, CALCON,
;       to the proper state and then outputs this pattern
;       to the port.  The effect is to turn all bits off for
;       the CALCON port except the required bit.  Since the
;       output buffer on that port is expected to invert the
;       signals the pattern is complemented before output.
;
```

```
COPCON:
        CALL    GENPAT  ; Generate the required pattern
                        ;  based on input values.
        OUT     CALCON  ; Output the control pattern to the
                        ;  Cal. Xfer Control port.
        RET             ; All done so return.
;
;-------------------------------------------------------------
        END
```

```
        NAME    IFIODR
        CSEG
;
;   ********************************************
;   *                                          *
;   *              IF I/O DRIVER               *
;   *                                          *
;   ********************************************
;
;       This package provides the basic input and
;   output drivers needed to communicate with the
;   IF Controller board.
;
;-------------------------------------------------
;
;       Public Declarations.
;
        PUBLIC  INIFIO, IFWRIT, IFREAD
;
;-------------------------------------------------
;
;   The following shows the I/O port designations
;   for all data lines, device select lines,  and
;   control lines used to communicate with the IF
;   Controller.
;
;      Port     Bit    Signal Description
;   ----------------------------------------------
;       E4      0-7   Data Line Bits 0 thru 7
;       E6       0    Select Bit 0 / Data I/O Control
;       E6      1-3   Select Bits 1 trhu 3
;       E6       4    E4 On-board Buffer I/O Control
;       E6       5    Reset Signal
;       E6       6    Secondary strobe/enable signal
;       E6       7    Primary strobe/enable signal
;
;-------------------------------------------------
;
;       Declare Control Words and Addresses.
;
INICMD  EQU  10000000B  ; Initial Command to I/O
                        ;  control device (8255):
                        ;  Port E4 - Mode 0 Output
                        ;  Port E5 - Mode 0 Output
                        ;  Port E6 - Mode 0 Output
;
IOCNTR  EQU   0E7H      ; Command Register Address for
                        ;  I/O Control Device (8255).
;
IFCOMC  EQU   0E6H      ; IF Communications Control
```

```
                        ;  Port Address.
;
APSCNT  EQU   0E5H      ; APS Control Port Address.
;
IFDATA  EQU   0E4H      ; IF Data Bus I/O Port Address.
;
RSTCMD  EQU   00100000B ; RESET Command for the IFC
                        ;  (sent thru IFCOMC).
;
;_____________________________________________________
;
;       INIFIO  -  Initialize Controller I/O Ports
;
; Inputs:       None.
; Ouputs:       None.
; Destroys:     A.
;
INIFIO:
        MVI   A,INICMD  ; Send the initial command
                        ;  word to the IF's I/O port
        OUT     IOCNTR  ;  controller (8255).
                        ;
        MVI   A,RSTCMD  ; Output RESET signal to IF
                        ;  Controller and disable
        OUT     IFCOMC  ;  all of its recivers and
                        ;  transmitters.
        RET             ; Return to calling routine.
;
;_____________________________________________________
;
;       IFGSEL  -  IF Group Select Group Routine.
;
; Function:     Send a 3-bit group select integer
;               to the IF board via the IF Comm.
;               Control port (IFCOMC).
; Input:        C - Integer in the range 0 to 7.
; Outputs:      None.
; Calls:        Nothing.
; Destroys:     A, H, L, and Flags.
;
IFGSEL:
        MVI L,00000110B ; Start with bit 3 of IFCOMC.
        MVI A,00000100B ; Initial bit mask set to get
                        ;  third bit of integer.
IFGS05:
        MOV     H,A     ; Save bit mask in H.
        ANA     C       ; Mask out one bit of integer.
        JZ      IFGS10  ; If bit was zero use the bit
                        ;  reset command.
        MVI     A,1     ; Else, the bit was one so use
```

```
                        ;  the bit set command.
IFGS10:
        XRI   00000001B ; Compliment the set/reset bit.
        ORA     L       ; Combine bit select no. with
                        ;  bit set/reset command.
        OUT     IOCNTR  ; Send Bit Set/Reset control
                        ;  word to the I/O Controller.
        DCR     L       ; Set IFCOMC bit select no. for
        DCR     L       ;  next lower bit.
        RZ              ; If no. more bits then return.
        MOV     A,H     ; Else, adjust bit mask to get
        RRC             ;  next lower bit of integer.
        JMP     IFGS05  ; Repeat till finished.
;
;_____________________________________________________
;
;       IFINEN  -  IF Input Enable Operation.
;
; Function:     Set required bits of the IF Comm.
;               Control Port (IFCOMC) to put IFC
;               into transmit mode and enable the
;               currently selected transmitter.
; Destroys:     A.
;
; No Inputs or Outputs.  Calls nothing.
;
IFINEN:
;
;       -  First set IFC in Transmit Mode.  -
;
        MVI A,00000001B ; Control word to set the IOC
                        ;  Bit (bit 0) of IFCOMC.
        OUT     IOCNTR  ; Send the bit set control
                        ;  word to the I/O Controller.
;
;       -  Then enable the selected transmitter.  -
;
        CALL    IFSEON  ; Use the IF Strobe/Enable set
                        ;  routine.
        RET             ; Return to calling routine.
;_____________________________________________________
;
;       IFSEON  -  IFC Strobe/Enable Set Routine.
;
; Function:     Turn on both the primary and secondary
;               Strobe/Enable Signals of the IFC.
; Destroys:     A.
;
; No Inputs or Outputs.  Calls nothing.
;
```

```
IFSEON:
        MVI A,00001101B ; Control word to set the PS/E
                        ;  Bit (bit 6) of IFCOMC.
        OUT     IOCNTR  ; Send the bit set control
                        ;  word to the I/O Controller.
        MVI A,00001111B ; Control word to set the SS/E
                        ;  Bit (bit 7) of IFCOMC.
        OUT     IOCNTR  ; Output it to the I/O Cntr.
                        ;
        NOP             ; Delay
        NOP             ;       a short
        NOP             ;                period
        NOP             ;                       of time.
        RET             ; Return to caller.
;
;_____________________________________________
;
;       IFSEOF  -  IFC Strobe/Enable Reset Routine.
;
; Function:     Turn off both the primary and secondary
;               Strobe/Enable Signals of the IFC.
; Destroys:     A.
;
; No Inputs or Outputs.  Calls nothing.
;
IFSEOF:
        MVI A,00001100B ; Control word to reset the PS/E
                        ;  Bit (bit 6) of IFCOMC.
        OUT     IOCNTR  ; Send the bit reset control
                        ;  word to the I/O Controller.
        MVI A,00001110B ; Control word to reset the SS/E
                        ;  Bit (bit 7) of IFCOMC.
        OUT     IOCNTR  ; Output it to the I/O Cntr.
                        ;
        RET             ; Return to caller.
;
;_____________________________________________
;
;       IFDBTX  -  IF Data Bus Tx Mode Operation.
;
; Function:     Set the IFC Data Bus into the Tx.
;               Mode.
; Destroys:     A and B.
;
; No Inputs or Outputs.  Calls nothing.
;
; Description:  The program first reads the status
;               of output port E5 to see what is
;               currently written to it.  Next the
;               I/O Controller is reprogrammed to
```

```
;               put port E4 in Mode 0 output oper.
;               (ports E5 & E6 remain in Mode 0
;               output operation).  Then the status
;               of port E5 is restored by sending out
;               the previously read data.
;
IFDBTX:
        IN      APSCNT  ; Input the status of port
                        ;  E5 (APS Control Port).
        MOV     B,A     ; Save this info. in B.
        MVI A,10000000B ; Output Mode Control word to
        OUT     IOCNTR  ;  the I/O Controller (8255).
        MOV     A,B     ; Get back APSCNT port data &
        OUT     APSCNT  ;  restore the data to the port.
        RET
;
;_____________________________________________________
;
;       IFDBRX  -  IF Data Bus Rx Mode Operation.
;
; Function:     Set the IFC Data Bus into the Rx.
;               Mode.
; Destroys:     A and H.
;
; No Inputs or Outputs.  Calls nothing.
;
; Description:  The program first reads the status
;               of output port E5 to see what is
;               currently written to it.  Next the
;               I/O Controller is reprogrammed to
;               put port E4 in Mode 0 input oper.
;               (ports E5 & E6 remain in Mode 0
;               output operation).  Then the status
;               of port E5 is restored by sending out
;               the previously read data.
;
IFDBRX:
        IN      APSCNT  ; Input the status of port
                        ;  E5 (APS Control Port).
        MOV     H,A     ; Save this info. in H.
        MVI A,10010000B ; Output Mode Control word to
        OUT     IOCNTR  ;  the I/O Controller (8255).
        MOV     A,H     ; Get back APSCNT port data &
        OUT     APSCNT  ;  restore the data to the port.
        RET
;
;_____________________________________________________
;
;       IFWRIT  -  IF Write Routine
;
```

```
; Function:     Output data to the selected IF
;               device.
; Inputs:       C - Destination Device Select Number.
;               E - Data Byte to be output.
; Outputs:      None.
; Destroys:     A, H, L, Flags.
; Calls:        IFDBTX, IFGSEL, IFSEON, & IFSEOF.
;
; Description:  The following sequence is performed
;               for a write operation to the IF
;               Controller -
;
;       1) Program the 8255 to set up the IF data port
;       (IFDATA) for output.  This inherently resets
;       the IF Communications Control Port which has
;       the following consequences:
;               a) Disable all IF Receivers (Rx) and
;               Transmitters,
;               b) Select Group 7 of the 8 IF Rx/Tx
;               pairs,
;               c) Set the I/O control bit for Output
;               which sets IF data transceiver in the
;               transmit mode and puts IFC into the
;               receive mode.
;
;       2)  Transmit Device Seclect Number to the IF
;       Controller.
;
;       3)  Output data byte to IFC
;
;       4)  Send strobe signal to IFC.
;
;
IFWRIT:
; 1)
        CALL    IFDBTX  ; Program Data Bus for Tx.
; 2)
        CALL    IFGSEL  ; Set the select bits of IFCOMC
                        ;  for the proper destination.
; 3)
        MOV     A,E     ; Place data on IF Data Bus.
        OUT     IFDATA  ;
; 4)
        CALL    IFSEON  ; Turn the strobe line on.
        CALL    IFSEOF  ; Turn the strobe off.
        RET             ; All done.
;
;_____________________________________________________
;
;       IFREAD  -  IF Read Routine.
```

```
;
; Function:     Input data from a selected source
;               device in the IF Controller.
; Inputs:       C - Source Device Select No.
; Outputs:      A - Data read from Source Device.
; Destroys:     A, H, L, and Flags.
; Calls:        IFDBRX, IFGSEL, IFINEN, & IFSEOF.
;
; Description:  The following sequence is performed
;               for a read operation from the IFC -
;
;       1) Program the 8255 to set up the IF data port
;       (IFDATA) for input.  This inherently resets
;       the IF Communications Control Port which has
;       the following consequences:
;               a) Disable all IF Receivers (Rx) and
;               Transmitters (Tx),
;               b) Select Group 7 of the 8 IF Rx/Tx
;               pairs,
;               c) Set the I/O control bit for Output
;               which sets IF data transceiver in the
;               transmit mode and puts IFC into the
;               receive mode.
;               d) Remove the reset signal that may
;               have been present on the IFCOMC port.
;               (bit 5 of IFCOMC).
;
;       2)  Send Source Device select no. to the IFC.
;
;       3)  Set the I/O Control Bit for Input (which sets
;       the IF data transceiver in the receive mode and
;       puts the IFC in the transmit mode) and send the
;       transmit enable signal to IFC.
;
;       4)  Read the data in from the IFC Data Bus.
;
;       5)  Remove the enable signal from the IFC.
;
;
IFREAD:
; 1)
        CALL    IFDBRX  ; Program the IF Data Bus for
                        ;  reception from the IFC.
; 2)
        CALL    IFGSEL  ; Transmit the source device
                        ;  select no. to the IFC.
; 3)
        CALL    IFINEN  ; Enable the selected IFC
                        ;  transmit device.
; 4)
```

```
        IN      IFDATA  ; Read the IFC Data bus.
        MOV     H,A     ; Save Data in H.
; 5)
        CALL    IFSEOF  ; Turn the IFC Enable off.
;
        MOV     A,H     ; Put data back in A and return.
        RET
;
;_____________________________________________________
;
        END
```

```
$TITLE('AIHTSK:  Analog Input Handler Task, CWK-82')

AIHTSK: DO;
/*
*********************************************************

Name:           AIHTSK - Analog Input Handler Task.
Priority:       130.
          Message------From/To-----Via-----Comments
Receives: AIREQM     Any Task    AIREQX  Request for
                                         AIH Services.
Sends:    AIREQM     Requesting  User    Response to
                       Task       Exch.  Task when done.

Pub. Procs. Used:  RQCXCH, RQWAIT, RQSEND

Function:  This task services requests for analog input
           from the iSBX-311 Multimodule Analog Input
           board.  The task will convert a single channel
           once or repetitively up to 256 times or scan a
           group of channels one time only.

Request:
        The requesting task must build a message in memory
in order to request the AIHTSK service.  The message must
be sent to AIHEXG.  The following must be passed in it:

      LENGTH:   14.
        TYPE:   AI$SNGL$TYPE = 100 for single channel conversion
                AI$SCAN$TYPE = 101 for multi-channel scan conv.
   HOME EXHG:   Not Used.
RESPONSE EXCH:  Address of user defined exchange to which
                AIHTSK will send the request message to
                when it has finished the requested service.
     CHANNEL:   Channel No. to be converted.  For TYPE=101
                this is the 1st channel of the group and
                others are sequential no.s greater than this.
                (One Byte.)
DATA POINTER:   Address of array that is to receive the data.
                The elements of the array should equal COUNT
                and each element is two bytes.  Only the lower
                twelve bits contain data (upper 4 are zero).
                (Two Bytes.)
       COUNT:   Number of times that a single channel is to be
                repetitively converted for TYPE=100 or number of
                channels to be converted for TYPE=101.
                (One Byte.)
      ACTUAL:   Number of data elements that were obtained by
                AIHTSK.  This value is returned by AIHTSK.  If
```

```
                not same as COUNT then a failure occured.
                (One Byte.)

*************************************************************
*/
$EJECT
/*
        > RMX/80 LITERALS
*/
$INCLUDE(:F2:EXCH.ELT)
$INCLUDE(:F2:MSG.ELT)
$INCLUDE(:F2:COMMON.ELT)
$INCLUDE(:F1:AIHTSK.ELT)
/*
        > EXTERNAL RMX/80 PROCS.
*/
        /* RQSEND AND RQWAIT */
$INCLUDE(:F2:SYNCH.EXT)
        /* RQCXCH */
$INCLUDE(:F2:OBJMAN.EXT)
/*
        > EXTERNAL RDADS PROCS.
*/
        /* COMB */
COMB:   PROCEDURE (HI$B,LO$B) ADDRESS EXTERNAL;
        DECLARE (HI$B,LO$B) BYTE;
    END COMB;
/*
        > EXCHANGES
*/
        DECLARE AIREQX EXCHANGE$DESCRIPTOR PUBLIC;
/*
        > LOCAL VARIABLES, CONSTANTS, AND LITERALS.
*/
        DECLARE SBX$BASE$ADDR LITERALLY '0C0H';  /* J5 on iSBC-80/24
*/
        DECLARE SBX$NEXT$ADDR LITERALLY '0C1H';
        DECLARE HI$DAT LITERALLY 'SBX$BASE$ADDR';
        DECLARE STATUS LITERALLY 'SBX$NEXT$ADDR';
        DECLARE LO$DAT LITERALLY 'SBX$NEXT$ADDR';
        DECLARE CHAN$SEL LITERALLY 'SBX$BASE$ADDR';
        DECLARE AIREQM$PNTR ADDRESS;
        DECLARE (HI$VAL,LO$VAL,I,CHAN) BYTE;
        DECLARE DATA$BUF$PNTR ADDRESS;
        DECLARE WHOLE$VAL ADDRESS;
        DECLARE DATA$BUF BASED DATA$BUF$PNTR (1) ADDRESS;
        DECLARE EOC$MASK LITERALLY '00000001B';
        DECLARE NOT$EOC LITERALLY '= EOC$MASK';
/*
        > MESSAGES
```

```
*/
        DECLARE AIREQM BASED AIREQM$PNTR AIHMSG$DESCRIPTOR;
/* */
$EJECT
AIHTSK: PROCEDURE PUBLIC;
/*
        > Start Up Initialization for AIHTSK.
*/
        HI$VAL = INPUT(HI$DAT); /* Reset the EOC/ bit of 311 */
/*
        > Main Processing Loop.
*/
        DO FOREVER;
            /* Wait indefinetly for request. */
            AIREQM$PNTR = RQWAIT(.AIREQX,0);
            AIREQM.ACTUAL = 0;
            DATA$BUF$PNTR = AIREQM.DATA$PNTR;
            CHAN = AIREQM.CHANNEL;
            I = 0;
            /* Check for proper message type. */
            IF AIREQM.TYPE = AI$SNGL$TYPE
            OR AIREQM.TYPE = AI$MULT$TYPE
            /* Perform repetitive conversions or multi-
                channel scan. */
            THEN DO WHILE I < AIREQM.COUNT AND CHAN < 16;
                /* Command SBX311 to convert a channel. */
                OUTPUT(CHAN$SEL) = CHAN;
                DO WHILE (INPUT(STATUS) AND EOC$MASK) NOT$EOC;
                     ;  /* Wait for EOC signal from 311. */
                   END;
                LO$VAL = INPUT(LO$DAT);  /* Input the data. */
                HI$VAL = INPUT(HI$DAT);  /* Inherent EOC/ reset. */
                WHOLE$VAL = COMB(HI$VAL,LO$VAL);
                /* Transfer data to the users buffer. */
                DATA$BUF(I) = SHR(WHOLE$VAL,4);
                AIREQM.ACTUAL = I:= I + 1;
                IF AIREQM.TYPE = AI$MULT$TYPE THEN CHAN = CHAN + 1;
              END;
            /* All through so send response to user. */
            CALL RQSEND(AIREQM.RESPONSE$EXCHANGE,AIREQM$PNTR);
          END;  /* * * End of Main Processing Loop. * * */
/* */
END AIHTSK;  /* End of Analog Input Handler Task Procedure. */
/* */
END;    /* End of AIHTSK Module. */
```

The REMOTE SENSING CENTER was established by authority of the Board of Directors of the Texas A&M University System on February 27, 1968. The CENTER is a consortium of four colleges of the University; Agriculture, Engineering, Geosciences, and Science. This unique organization concentrates on the development and utilization of remote sensing techniques and technology for a broad range of applications to the betterment of mankind.