AD-A091 630 PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING F/G 15/3
A MICRO-COMPUTER COMPUTATIONAL UNIT FOR AN IR-CCD INTRUSION DET--ETC(U)
OCT 80 T W GOEDDEL, W T WILSON, S C BASS F30602-75-C-0082
UNCLASSIFIED RADC-TR-80-308 NL

1 of 2
AD
A091630

[illegible]

[illegible]

APPROVED: [illegible]

[illegible]

APPROVED: [illegible]

[illegible]

FOR THE COMMANDER: [illegible]

[illegible]

SUBJECT TO EXPORT CONTROL LAWS

[illegible]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

18 19

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-80-308 ✓

2. GOVT ACCESSION NO.: AD-A091630

3. RECIPIENT'S CATALOG NUMBER:

6

4. TITLE (and Subtitle): A MICRO-COMPUTER COMPUTATIONAL UNIT FOR AN IR-CCD INTRUSION DETECTION SYSTEM.

9

5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report,

6. PERFORMING ORG. REPORT NUMBER: N/A

10

7. AUTHOR(s): T.W. Goeddel, W.T. Wilson, S.C. Bass

15

8. CONTRACT OR GRANT NUMBER(s): F30602-75-C-0082 ✓

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Purdue University, School of Electrical Engineering ✓, West Lafayette IN 47907

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63714F, 681E00P1

16 17 00

11. CONTROLLING OFFICE NAME AND ADDRESS: Deputy for Electronic Technology (ESE), Hanscom AFB MA 01731

11

12. REPORT DATE: Oct~~ober 1980~~

13. NUMBER OF PAGES: 184

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same

12 170

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Lyn H. Skolnik (ESE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
IRCCD
Physical security
Schottky array

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of the BISS (Base Installation Security System) program is the development of sensors for physical security systems. One technique under investigation within this program involves the focusing of infrared radiation, derived from a distant field of vision onto an integrated linear array of 256 platinum silicide Schottky barrier detectors. Over a period of time (called the "stare time"), charge packets, of a size proportional to the infrared intensity, develop within each detector. At

next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

292000 [illegible]

the end of the stare time, all 256 charge packets are broadside loaded into a CCD shift register (of length 256) also integrated onto the detector chip. The CCD register may then be clocked to shift out these analog samples for processing by a small local computer. The hardware and software development of such a signal processing computer is the purpose of the effort discussed in this report.

ABSTRACT

The purpose of the BISS (Base Installation Security System) program is the development of sensors for physical security systems. One technique under investigation within this program involves the focusing of infared radiation, derived from a distant field of vision onto an integrated linear array of 256 platinum silicide Schottky barrier detectors. Over a period of time (called the "stare time"), charge packets, of a size proportional to the infrared intensity, develop within each detector. At the end of the stare time, all 256 charge packets are broadside loaded into a CCD shift register (of length 256) also integrated onto the detector chip. The CCD register may then be clocked to shift out these analog samples for processing by a small local computer. The hardware and software development of such a signal processing computer is the purpose of the effort discussed in this report.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EVALUATION

The ability of the RADC/ESE IRCCD fence system to discriminate between targets and false alarm sources is dependent on the performance of its signal processor. Under this effort Purdue University has produced the hardware and software models for an advanced signal processor. In the laboratory this processor has met or exceeded all its design goals.

[illegible]
LYN H. SKOLNIK
Project Engineer

I. INTRODUCTION

A. Purpose of Overall System

The purpose of the BISS (Base Installation Security System) program is the development of sensors for physical security systems. One technique under investigation within this program involves the focusing of infrared radiation, derived from a distant field of view, onto an integrated linear array of 256 platinum silicide Schottky barrier detectors. Over a period of time (called the "stare time"), charge packets, of a size proportional to the infared intensity, develop within each detector. At the end of the stare time, all 256 charge packets are broadside loaded into a CCD shift register (of length 256) also integrated onto the detector chip. The CCD register may then be clocked to shift out these analog samples for processing by a small local computer. The hardware and software development of such a signal processing computer is the purpose of the effort discussed in this report.

B. The Computational Unit

Physically, the unit developed under this contract is a PDP-11/03 computer with certain optional boards tied to the bus. See Figure 1.1. These include 8k words (16 bits) of PROM (programmable read-only memory) for program storage, a DMA (direct memory access) board for highspeed inputting of CCD samples into RAM, and a custom-built interface board for scaling, digital conversion, and simple pre-processing of these samples.

With the exception of the latter "custom" board, all these components are standard items manufactured by Digital Equipment Corporation (DEC). These are documented in [1] together with Appendix Section A.1.

The custom interface board, detailed in Section II.B., receives the serially-shifted contents of the detector's CCD array, together with two digital pulse timing signals: START PACKET and START CONVERT. The former is required to appear immediately prior to the CCD's outputting of the first analog sample in the array of 256. The latter timing signal executes a positive-going transition to accompany each incoming CCD sample. This transition immediately initiates an A/D conversion of its corresponding analog (CCD) sample. Hence this pulse should not be transmitted until the analog data is certain to have settled at the custom interface input.

Two digital control signals, leaving the custom interface, inform the detector electronics of which <u>one of four</u> stare times are to be used by the detector array. As the overall dynamic range of the detector charge packets shrink or increase (with changing ambient temperature conditions), the microprocessor will instruct the detector electronics to change its operating stare time by a factor of two.

A computational algorithm for the automatic signaling of "intruder targets," given the IR-CCD data, was developed in [2]. A flow chart of the

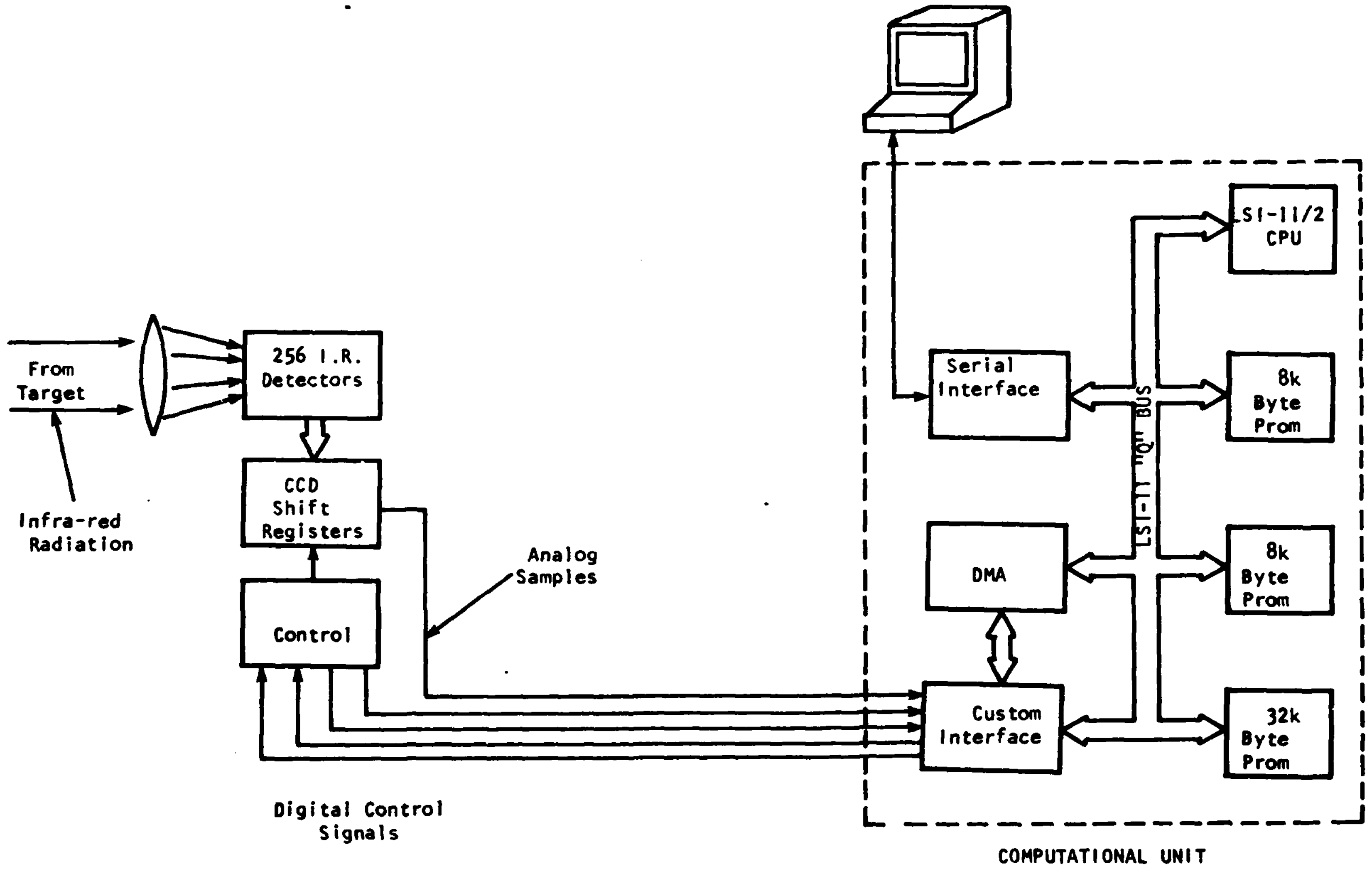
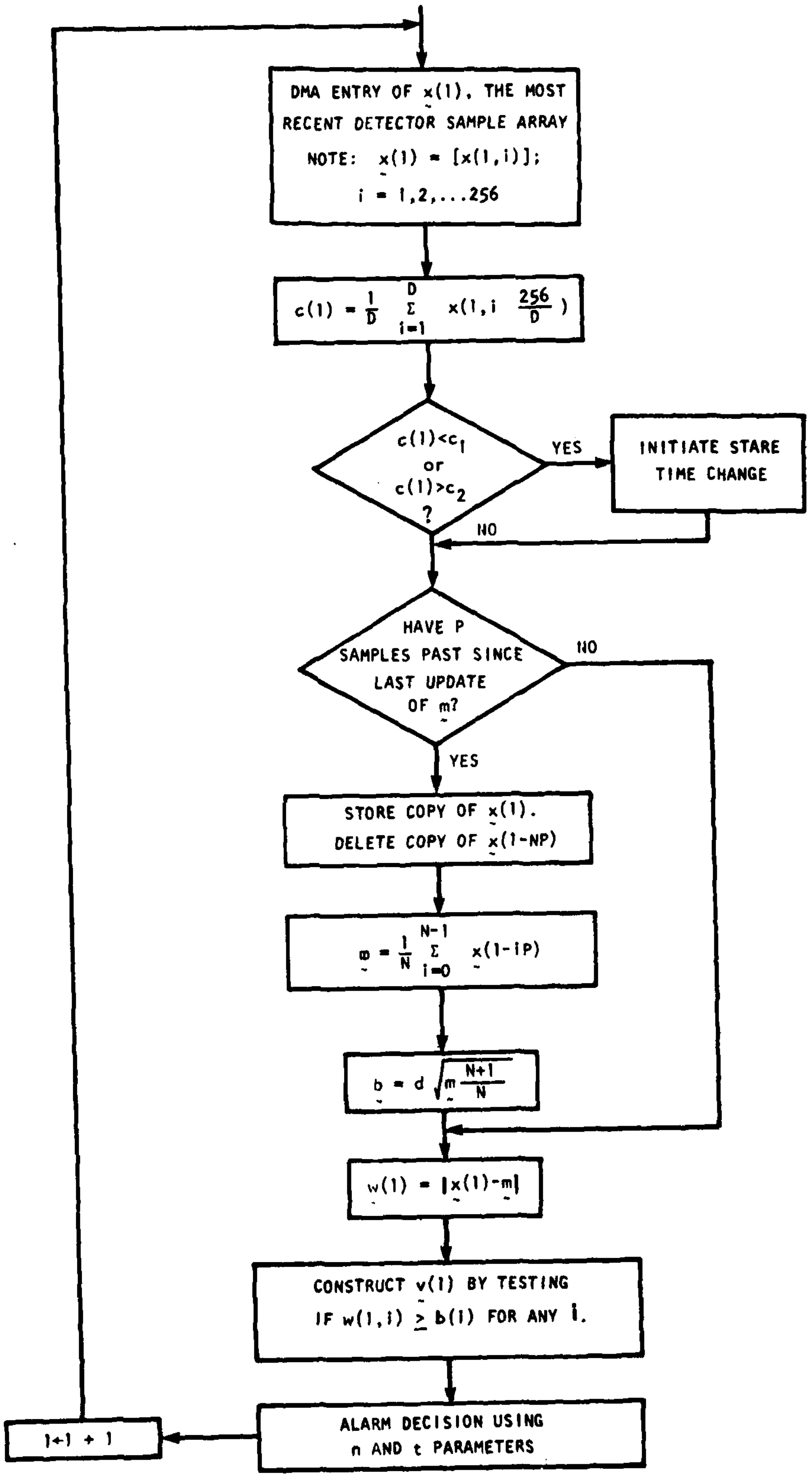


Figure 1.1 System level depiction of the computational unit described in this report together with its connections with the IR-CCD detector system.

calculations called for in [2] appears in Figure 1.2. The software developed under this effort carries out these calculations using the parameters D, C_1, C_2, P, N, d, n, and t. These parameters are adjustable by the user through terminal interaction. See Section I.E.

The software also includes a number of maintenance routines for testing and fault isolation on the custom, DMA, PROM, RAM, and serial interface circuit boards. See Section IV.

The software was written in PDP-11 assembly language and "C", and compiled under the UNIX operating system running on a PDP-11/70.



DMA ENTRY OF x(l), THE MOST RECENT DETECTOR SAMPLE ARRAY
NOTE: x(l) = [x(l,i)]; i = 1,2,...256
$c(l) = \frac{1}{D} \sum_{i=1}^{D} x(l, i\frac{256}{D})$
$c(l) < c_1$ or $c(l) > c_2$?
YES
INITIATE STARE TIME CHANGE
NO
HAVE P SAMPLES PAST SINCE LAST UPDATE OF m?
NO
YES
STORE COPY OF x(l). DELETE COPY OF x(l-NP)
$m = \frac{1}{N} \sum_{i=0}^{N-1} x(l-iP)$
$b = d\sqrt{m\frac{N+1}{N}}$
$w(l) = |x(l)-m|$
CONSTRUCT v(l) BY TESTING IF $w(l,i) \geq b(l)$ FOR ANY i.
ALARM DECISION USING n AND t PARAMETERS
$l \leftarrow l + 1$

Fig. 1.2 Flow diagram of the major calculations involved in a single cycle of the computational unit software.

C. Start-Up Procedures for IR-CCD Intrusion Detection System

The basic start-up procedures for the IR-CCD Intrusion Detector System will be described here. It is assumed that all connections to the IR-CCD interface have been properly made as discussed in the hardware section, and that the serial interface is connected to a Tektronix 4024 terminal set at 9600 baud.

The power switch for the PDP-11/03 minicomputer is located on the right hand side (when viewing the 11/03 from the front) of the back face. When this switch is moved to the on position (up), the 11/03 receives A.C. power and begins executing a built-in terminal handling routine known as "ODT." The ODT mode is described in some detail in [1], but briefly, it allows the user to control the Load Address, Deposit, Examine, Continue, and Start/Halt functions of the microprocessor. When in ODT, the user is prompted for a command with the "@" symbol. Whenever it is desired to begin execution of the IR-CCD Intrusion Detection program, the command "20000G" must be sent to ODT. This may be accomplished by one of two methods. If starting the routine from a "cold" status (both the microprocessor and the Tektronix 4024 have just been brought up from a powered down condition), it is necessary to type the "20000G" command after receiving the "@" ODT prompt. The second method is effective only when the Tektronix 4024 terminal has not been powered down since the program was last executed. In this case, the "PT" key on the far right hand side of the 4024 keyboard will have been programmed to execute the "20000G" sequence automatically. Hence, it is only necessary to depress the "PT" key to begin program execution.

To halt the routine, it is necessary to send a break command to the 11/03. On the Tektronixs 4024 terminal, this is accomplished by hitting the "BREAK" key rapidly twice in succession. The processor should return to ODT mode with the "@" prompt appearing on the screen.

The start-up procedure for the IR-CCD Intrusion Detection System may be summed up as follows:

(1) Connect Tektronixs 4024 terminal, set at 9600 baud, to serial interface on PDP-11/03 processor.

(2) Connect IR-CCD analog data and timing signal lines to IR-CCD interface board.

(3) Power up terminal and PDP-11/03.

(4) When ODT prompt (@) is received, issue "20000G" sequence by either:

(a) typing 20000G.

(b) depressing terminal "PT" key (if 4024 has not been turned off since last execution).

D. Description of Parameter Display Frame

When the start-up procedure outlined in the previous section is performed, the routine begins execution and places on the screen a table listing the initial default parameter values and a brief description of the meaning of each parameter. Immediately below this parameter display table is a box containing information about the status of the alarm. More will be said about this box later. At the bottom of the screen, two lines should appear informing the user of how to change a parameter value and how to switch the display to the "additional features" list frame. This initial frame is shown in Figure 1.3.

To change a parameter, the user should type the mnemonic for that parameter (which is listed in the first column of the parameter display table on the terminal screen) followed by a carriage return. For parameters wherein no ambiguity will result if upper or lower case symbols are typed, either case may be used, i.e., the "P" parameter. If ambiguity will result, then the exact mnemonic must be typed, i.e., the "n" or "N" parameters. If errors are made in typing, there are three keys which are programmed to be "delete character" keys and three keys which are programmed to be "delete line" keys. The delete character keys are:

(i)	DEL CHAR	(top row of 4024 keyboard)
(ii)	CTRL-H	(type H key while depressing "CTRL" key)
(iii)	#	(upper case 3)

while the delete line keys are:

(i)	DEL LINE	(top row of 4024 keyboard)
(ii)	CTRL-X	(type X key while depressing "CTRL" key)
(iii)	@	(upper case 2).

The delete character keys backspace one character while the delete line keys

```
                    IRCCD INTRUSION DETECTOR ALARM

 ____________________________________________________________________________
| SYMBOL |  VALUE  | MEANING                                                 |
|   D    |   16    | Number of cells monitored for stare time update.        |
|        |         |                                                         |
|   N    |   32    | Number of samples used in background time average.      |
|        |         |                                                         |
|   P    |   12    | Separation of samples used in background time average.  |
|        |         |                                                         |
|   d    |  6.17   | Detection threshold scale factor.                       |
|        |         |                                                         |
|   n    |   12    | 3 * memory interval length used in alarm decision.      |
|        |         |                                                         |
|   t    |    8    | Number of space-time threshold violations to cause alarm|
|        |         |                                                         |
|   C2   |   480   | Upper threshold used to initiate stare time decrease.   |
|        |         |                                                         |
|   C1   |   230   | Lower threshold used to initiate stare time increase.   |
|________|_________|_________________________________________________________|

                        *********************
                        *                   *
                        *   ALARM DISABLED  *
                        *                   *
                        *********************

        To see a list of other features, type "L".

        To change a parameter, type symbol shown above.
```

Figure 1.3 Initial parameter display frame.

erase the entire line that has been typed.

When the mnemonic for a parameter is typed, one of two things will happen. If the mnemonic is not a valid one, an error message will be displayed and the prompt for changing a parameter will be repeated. If the mnemonic is a valid one, the prompt for changing a parameter will be replaced by a line indicating the possible values that that parameter may be set to. An example of this is shown in Figure 1.4.

At this point, the user has three options. He may (i) type in the desired new value of the parameter followed by a carriage return, or (ii) type in the mnemonic for a different parameter followed by a carriage return, or (iii) depressed a carriage return. If a new mnemonic is typed in, the list of possible values of the original parameter will be replaced by a list of possible values of this new parameter. If a carriage return is depressed, the display will return to the original frame, and no parameter change will take place. Thus, if the user decides after examining the list of possible parameter values that he no longer desires to change that parameter, he may pass over the actual changing operation.

When a new parameter value is typed, one of several things may occur. If the new value is not one of those allowed, an error message is displayed and the list of possible parameter values is repeated. If the new value is an allowed one for that parameter, the old value listed in the second column of the parameter display table is replaced by the new value. When a parameter that may take on only integer values is changed (any one except the "d" parameter), the value typed is rounded off to the nearest integer. If the "d" parameter is changed, the value typed is rounded off to two decimal places. These rounding operations are carried out before determining if the new parameter value is valid. After a parameter change has taken place, the

```
                       IRCCD INTRUSION DETECTOR ALARM
 ______________________________________________________________________________
| SYMBOL  | VALUE   | MEANING                                                  |
|   D     |  16     | Number of cells monitored for stare time update.         |
|         |         |                                                          |
|   N     |  32     | Number of samples used in background time average.       |
|         |         |                                                          |
|   P     |  12     | Separation of samples used in background time average.   |
|         |         |                                                          |
|   d     |  6.17   | Detection threshold scale factor.                        |
|         |         |                                                          |
|   n     |  12     | 3 * memory interval length used in alarm decision.       |
|         |         |                                                          |
|   t     |   8     | Number of space-time threshold violations to cause alarm|
|         |         |                                                          |
|   C2    |  480    | Upper threshold used to initiate stare time decrease.    |
|         |         |                                                          |
|   C1    |  230    | Lower threshold used to initiate stare time increase.    |
|_________|_________|__________________________________________________________|

                          *********************
                          *                   *
                          *   ALARM DISABLED  *
                          *                   *
                          *********************

              To see a list of other features, type "L".

              Possible values for N are 2<=N<=40. N=?
```

Figure 1.4 Frame showing typical prompt for changing a parameter.

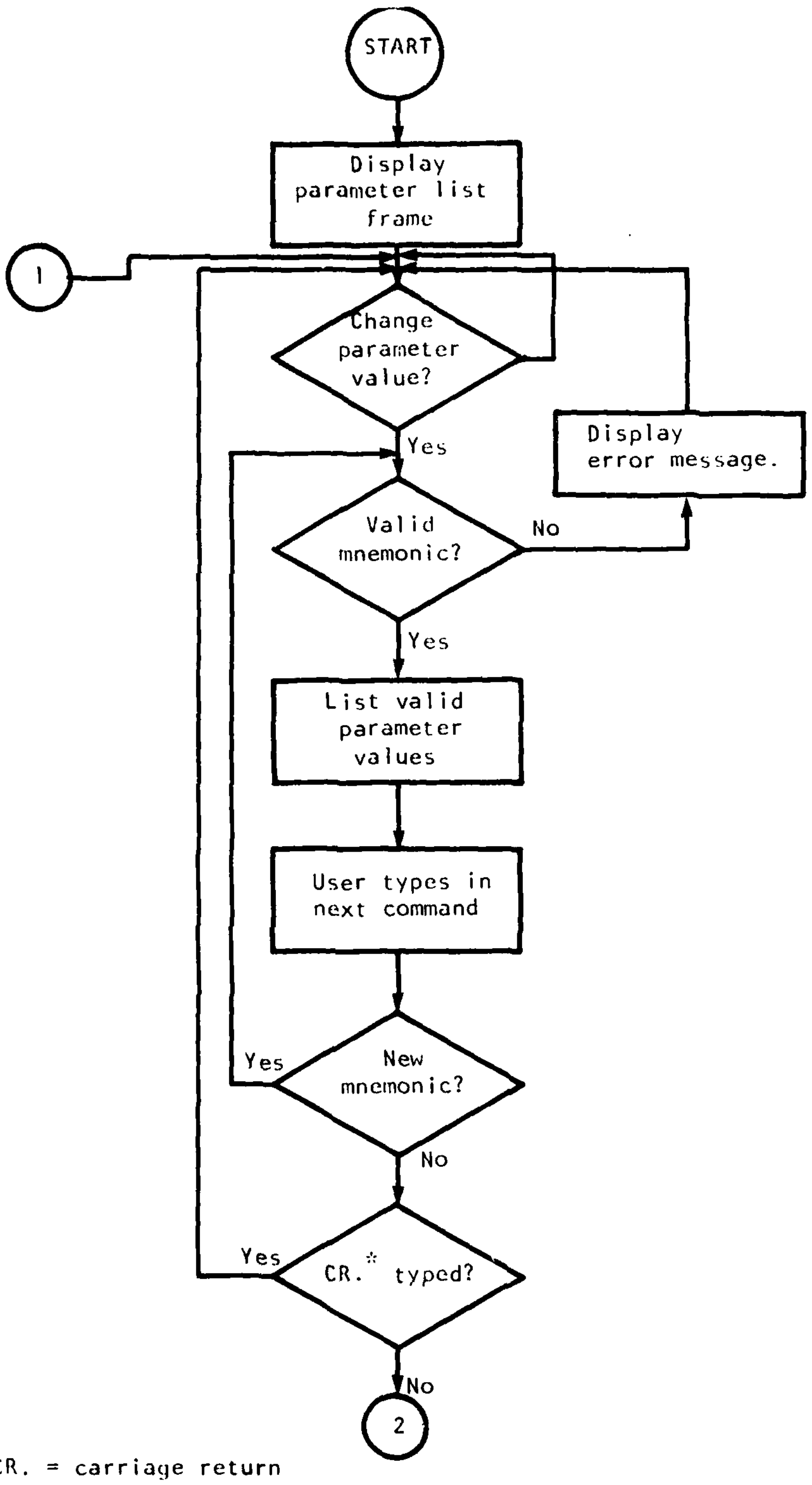
messages at the bottom of the screen are restored to their original state, informing the user of how to change a parameter value. All allowed parameter values and their default values are listed in Table 1.1. Figure 1.5 shows a flow chart illustrating the entire parameter changing operation.

The last section of the parameter display frame which has not been discussed in any detail yet is the alarm status box located in the lower center section of the screen. The information in this box tells the user which one of three possible conditions exists. These conditions are (i) that the alarm is disabled, (ii) that no target is currently being detected, or (iii) that a target is currently being detected. The alarm is disabled any time there is an insufficient amount of data based on the current parameter .alues to ensure a valid target decision. This condition will occur in several different situations. When the routine is first started, it is necessary to compute a background time average which extends N*P packets into the past. Thus the alarm is disabled for N*P packets while this initial background time average is computed. With a time between packets of about 100 msec, the time the alarm is disabled (assuming default parameter values) is approximately 38 seconds. A similar situation occurs whenever the values of the N or P parameters are changed. When the parameter n is changed, it is necessary that threshold violation information be known for the n/3 most recent packets. Therefore the alarm must be disabled for n/3 packets before a valid target decision can be made.

When the alarm is enabled, either the no target detected or target detected message is displayed in the alarm status box. The message is updated any time the computations determine a change is in order or when the alarm is disabled or enabled as explained previously.

Parameter	Default Value	Possible Values
D	16	16 or 32
N	32	$2 \leq N \leq 40$
P	12	$10 \leq P \leq 256$
d	6.17	$0 < d < 10$
n	12	6, 9, 12, or 15, $n \geq t$
t	8	$3 \leq t \leq n$
C2	480	$2 * C1 < C2 \leq 511$
C1	230	$0 \leq C1 < \frac{1}{2} * C2$

Table 1.1 Possible parameter values and their defaults.



*CR. = carriage return

Figure 1.5 Flow diagram of parameter changing operation.

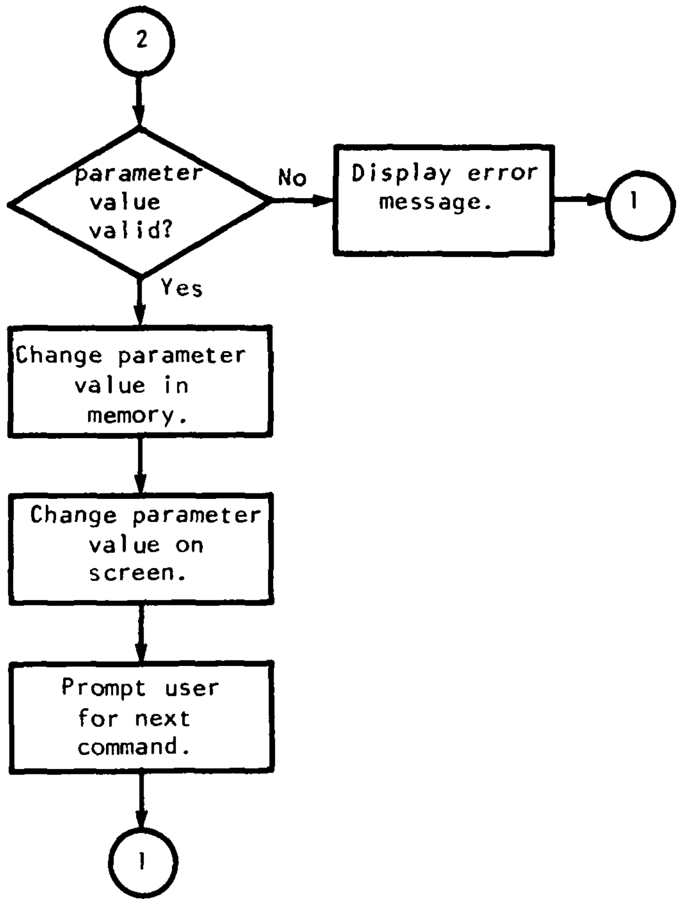

2
parameter value valid?
No
Display error message.
1
Yes
Change parameter value in memory.
Change parameter value on screen.
Prompt user for next command.
1

Figure 1.5 continued.

In addition to this visual indication of target detection information, there is an audible alarm signal which sounds every time a target is detected by the computational routine provided the following conditions are met:

(i) The alarm is not disabled.

(ii) The audible alarm feature is enabled.

The audible alarm is in the enabled state when the routine is first started. The status of the audible alarm may be changed, as will be described in the next section.

This completes the discussion of the parameter display frame and the procedure for changing parameter values. For a more detailed discussion of the meanings of the various parameters, the reader is referred to [2].

E. Description of "Additional Features" Frame

In addition to performing the basic intrusion detection function, the IR-CCD Intrusion Detection System has several self-diagnostic and testing features which are accessed via an "additional features" frame displayed on the terminal. This frame may be displayed by one of two methods. When the terminal is displaying the normal parameter display frame, and the routine is waiting for a parameter mnemonic to be typed, the additional features list will be displayed if (i) an "L" (upper or lower case) is typed followed by a carriage return, or (ii) a carriage return by itself is typed. Similarly, when the additional features frame is being displayed and the routine is waiting for a feature to be selected, the display will switch to the parameter display frame if (i) an "8" is typed or, (ii) if a carriage return is typed. Thus, the user may toggle between the two frames by repeatedly hitting the carriage return key. The additional features frame is shown in Figure 1.6.

```
                IRCCD INTRUSION DETECTOR ALARM

                    Additional Features:

        1) Disable audible alarm.
        2) Display output of interface A/D converter.
        3) Programable digital shift circuitry test.
        4) DMA channel test.
        5) RAM integrity test.
        6) PROM checksum test.
        7) Serial interface/terminal test.
        8) Do nothing - return to parameter display.




                    *********************
                    *                   *
                    *   ALARM DISABLED  *
                    *                   *
                    *********************


           Please type number of desired feature.
```

Figure 1.6 "Additional features" frame.

A particular "additional" feature is selected by typing the number shown on the display corresponding to that feature, followed by a carriage return. The delete character and delete line keys are the same as for the parameter display frame. If an error is made in typing a feature selection number, an appropriate error message is displayed at the bottom of the screen. If a valid feature is selected, one of two possibilities will occur. If the audible alarm enable/disable feature is selected, the audible alarm status will be changed and the routine will switch back to the original parameter display frame. However, when one of the self-diagnostic tests is selected, the screen is blanked and a brief description of the selected test along with any special instructions are displayed. At this point, except for the DMA channel test, the user is prompted to hit a carriage return to go ahead and execute the test or any other ascii character to skip the test and return to the additional features frame. Since it is necessary for the user to halt the 11/03, power down, and insert the DMA maintenance cable in order to perform the DMA test, when this feature is selected the user is prompted to hit a carriage return to halt the routine, or any other ascii key to skip the test and return to the additional features frame.

The actual operation of the self-diagnostic test routines will be covered in the section on maintenance, and is not dealt with here. The various test routines are listed in Table 1.2 along with a brief description of each test. Once a test is completed and the appropriate exit command given, the routine reinitializes itself to the default parameter values and begins computations, as was done at start-up time. To enter another test routine, the user must first toggle over to the additional features frame (since restarting returned the screen to the parameter display frame) and go through the feature selection procedure discussed previously.

Test	Description
A to D converter	Displays 12 bit output of A to D converter on screen in both binary and decimal.
Programmable Digital Shift Logic	Cycles through all possible function 1 and function 2 line values and displays output of programmable digital shift logic on screen in both binary and decimal.
DMA	Tests various functions of the DMA board through use of DMA maintenance cable.
RAM Integrity	Each word of RAM is tested for its Read/Write integrity.
PROM	Checksums are computed for both the lower and upper 4k words of PROM.
Terminal/Serial Interface	The terminal is tested via the standard Tektronixs 4024 'test' command and then the serial interface is tested by sending unique ascii character patterns to the terminal.

Table 1.2 Brief descriptions of software maintenance routines.

The alarm status box, which contains the visual target detection information, is serviced throughout the additional feature selection process until one of the self-diagnostic tests is actually begun. Target detection computations cease once the user decides to execute the test (after viewing the test description frame).

A flow chart illustrating the additional feature selection process is shown in Figure 1.7. This completes the discussion of the additional features frame and the additional feature selection process.

F. Verification of System Operation

The setup shown in Figure 1.8 was used to test the operation of the IR-CCD Intrusion Detection System during its development. A software routine written for the PDP-11/70 was used to generate files containing up to 256 packets of data samples. These files were then down-loaded to a floppy disk attached to a PDP-11/10. The 11/10 was then instructed to send these samples to a 12 bit DAC in real time under the control of external timing circuitry. After conversion, this analog signal was sent to the PDP 11/03 custom interface board. The external timing circuitry also supplied the "start packet" and "start convert" signals for the 11/03.

The procedure for testing involved generating known packets of data, sending them to the IR-CCD system as described above, and then observing the target/no target decision responses. This was done for several different patterns of packets and a large number of possible parameter combinations. All test results agreed with those expected.

The speed of computations was also tested by applying only the clock signals (analog input grounded). The nominal time between packets was specified to be 100msec. Experimental results indicate that the unit will actually run at up to twice that speed for worst case parameter combinations

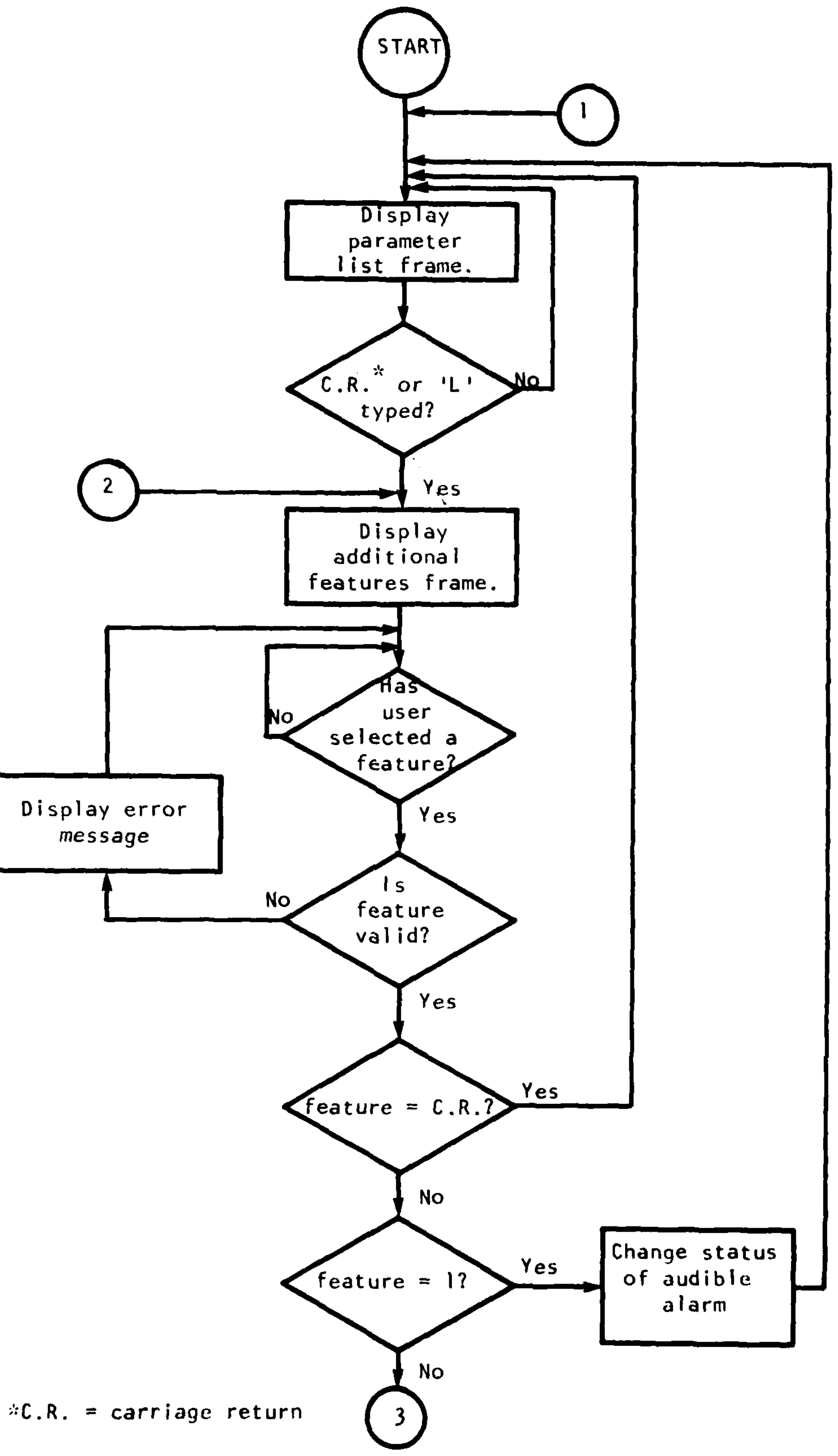


Figure 1.7 Flow diagram illustrating additional feature selection process.

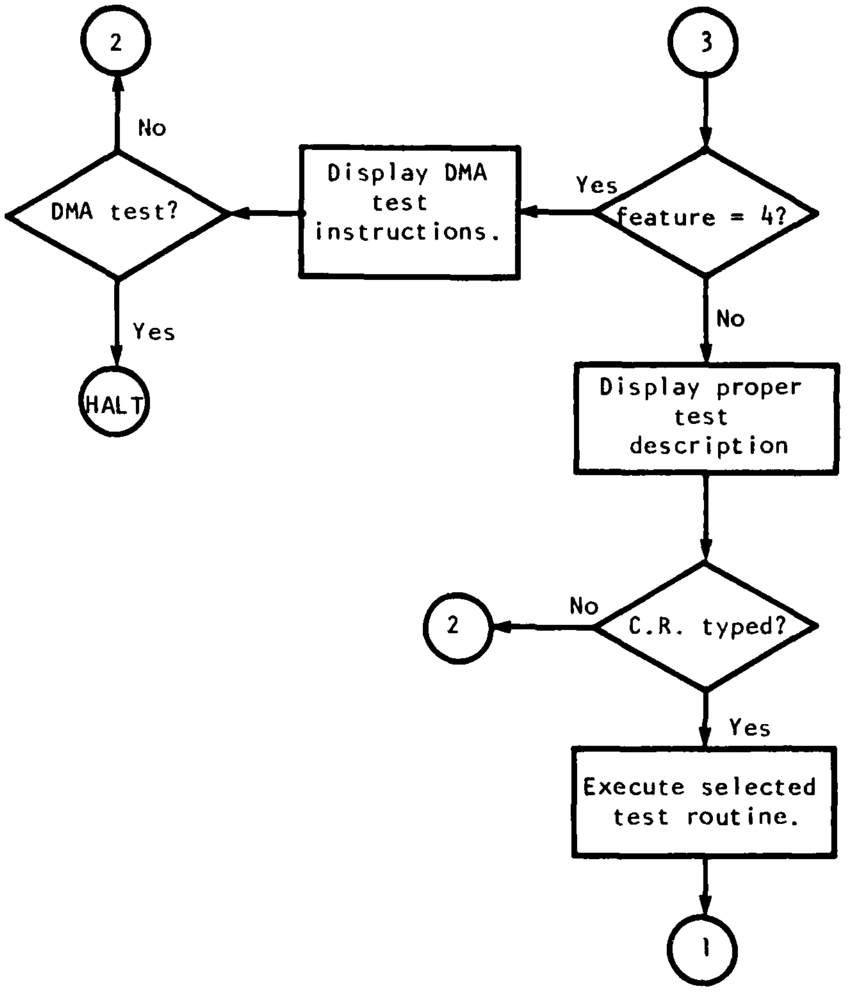

2
3
No
Display DMA test instructions.
Yes
DMA test?
feature = 4?
Yes
No
HALT
Display proper test description
No
2
C.R. typed?
Yes
Execute selected test routine.
1

Figure 1.7 continued.

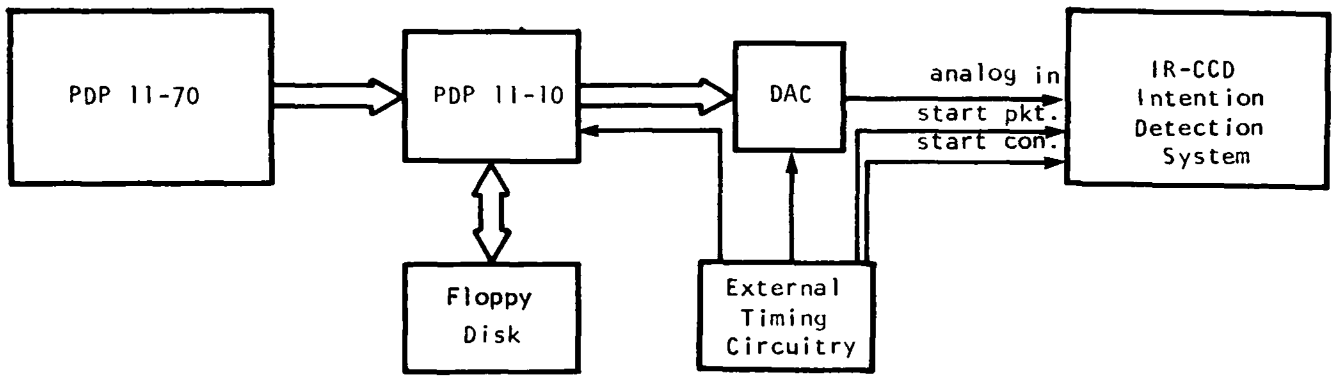


Figure 1.8 Setup used to verify system operation.

and at about 40msec (2 1/2 times faster) for the default parameter values.

II. Hardware

A. Overview of Hardware Operation

Physically, the unit developed under this contract is a PDP-11/03 computer with certain optional boards tied to the bus. See Figure 1.1 for a system level depiction of the computational unit. These boards include 8K words (16 bit) of PROM for program storage, a 16K word RAM, a serial interface for terminal communications, a DMA board for high-speed reception of CCD samples into RAM, and a custom-built interface board for scaling, digital conversion, and simple pre-processing of these samples. Figure 4.1 indicates board location in the PDP-11/03.

With the exception of the latter "custom" board, all these components are standard items manufactured by Digital Equipment Corporation (DEC). These are documented in [1] together with Appendix Section A.

Figure 2.1 depicts a functional level drawing of the IR-CCD interface board. The left side of the drawing represents connections to the IR-CCD device, while the right side represents connections to the LSI-11 computational unit.

The positive analog input voltage from the IR-CCD device is sampled and held constant by a Burr-Brown SHM60 high-speed sample-and-hold unit. This device has a 1 usec acquisition time, 12 nsec aperature time, and a variable gain. The gain can be varied between +1 and +1000 by use of a 10K ohm "trim pot" located near the device. The held voltage is converted to a 12-bit complementary 2's complement number using a Burr-Brown ADC85C-12 analog-to-digital converter. In complementary 2's complement (00....00) represents positive full scale while (11....11) represents zero. Since both the SHM60 and ADC85C-12 devices require ± 15 volts, as well as +5 volts, a Burr-Brown DC/DC 546 voltage converter is required. Positive 5 volts from the

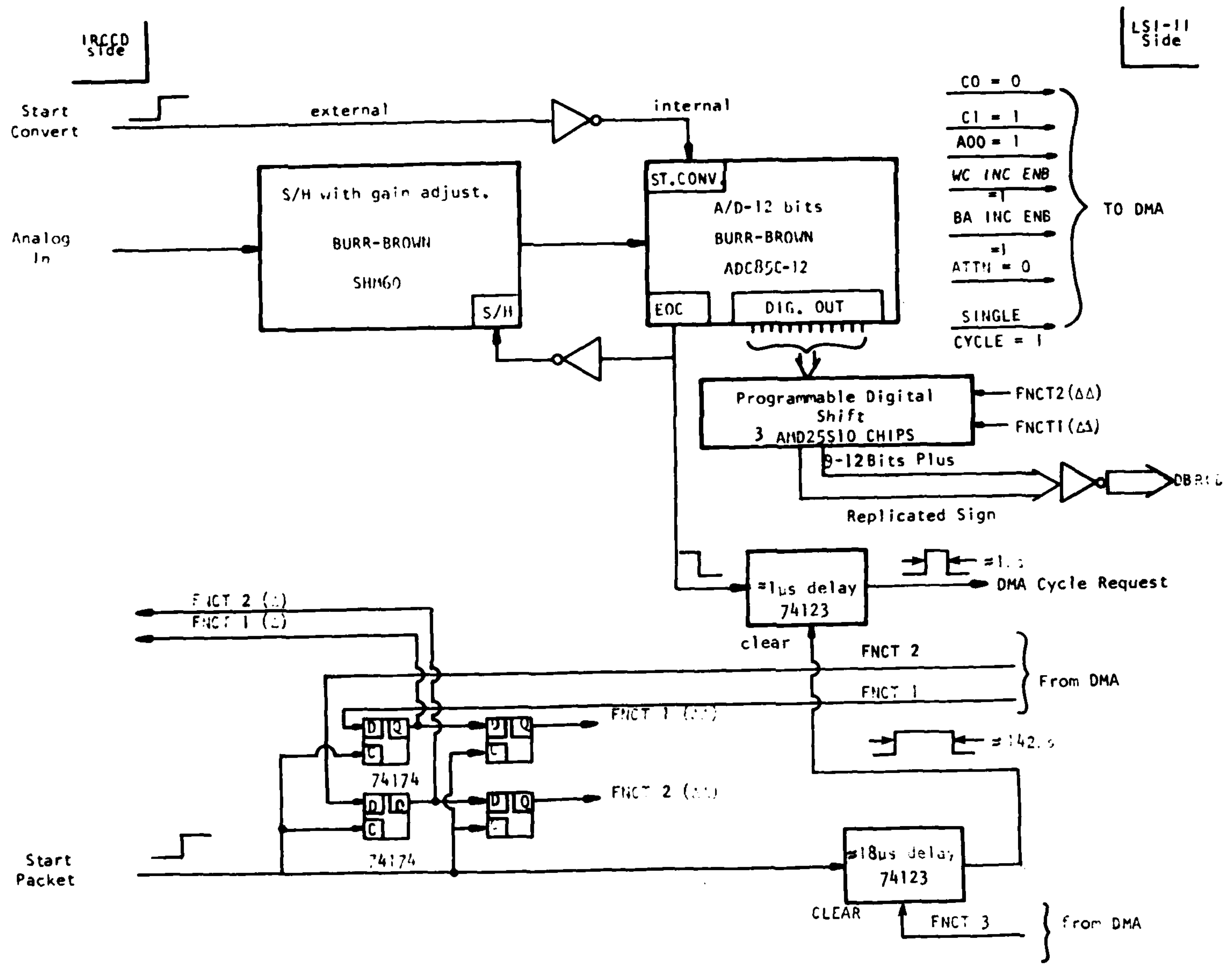


Figure 2.1 Functional level drawing of IR-CCD interface board.

LSI-11/03 bus is converted to ±15 volts by this device. Specifications for these three devices are given in Appendix Section A.

The remaining IR-CCD interface board components are resistors, capacitors, and TTL logic devices. The +5 volt supply pin of each TTL chip is bypassed to its ground pin using a .01 uf ceramic capacitor located on the underside of the interface board. The TTL chips include three SN 74123 (one-shot multivibrators), one SN 74174 (hex D-type flip-flop), two SN 7404 (hex inverters), one SN 7414 (hex Schmidt-Trigger inverter), and three AMD 25S10. Except for the latter, all are standard TTL devices with specifications given in any TTL data book. The AMD 25S10 is a 4-bit shifter. This device has the ability to shift four bits of parallel data 0, 1, 2, or 3 places. The AMD 25S10 has two select lines that are decoded internally to determine the number of places the data is shifted. Specifications for this device are also given in Appendix Section A.

B. Step-by-Step Discussion of IR-CCD Interface Board Operation

Figure 2.2 is a timing diagram that will aid one's understanding of the following step-by-step explanation of the IRRCD Interface operation.

We start this discussion with the arrival of a start packet pulse from the IR-CCD device heralding the arrival of a pa.ket of 256 CCD voltages. The start packet pulse is sent through two successive Schmidt-Trigger inverters to reduce any noise on the start packet line which might be misinterpreted as another start packet. The positive-going threshold is 1.7 volts and the negative-going threshold is .9 volts. Therefore, only glitches larger than 1.7 volts will cause an erroneous start packet to occur. From this point on, "start packet" will refer to this conditioned signal.

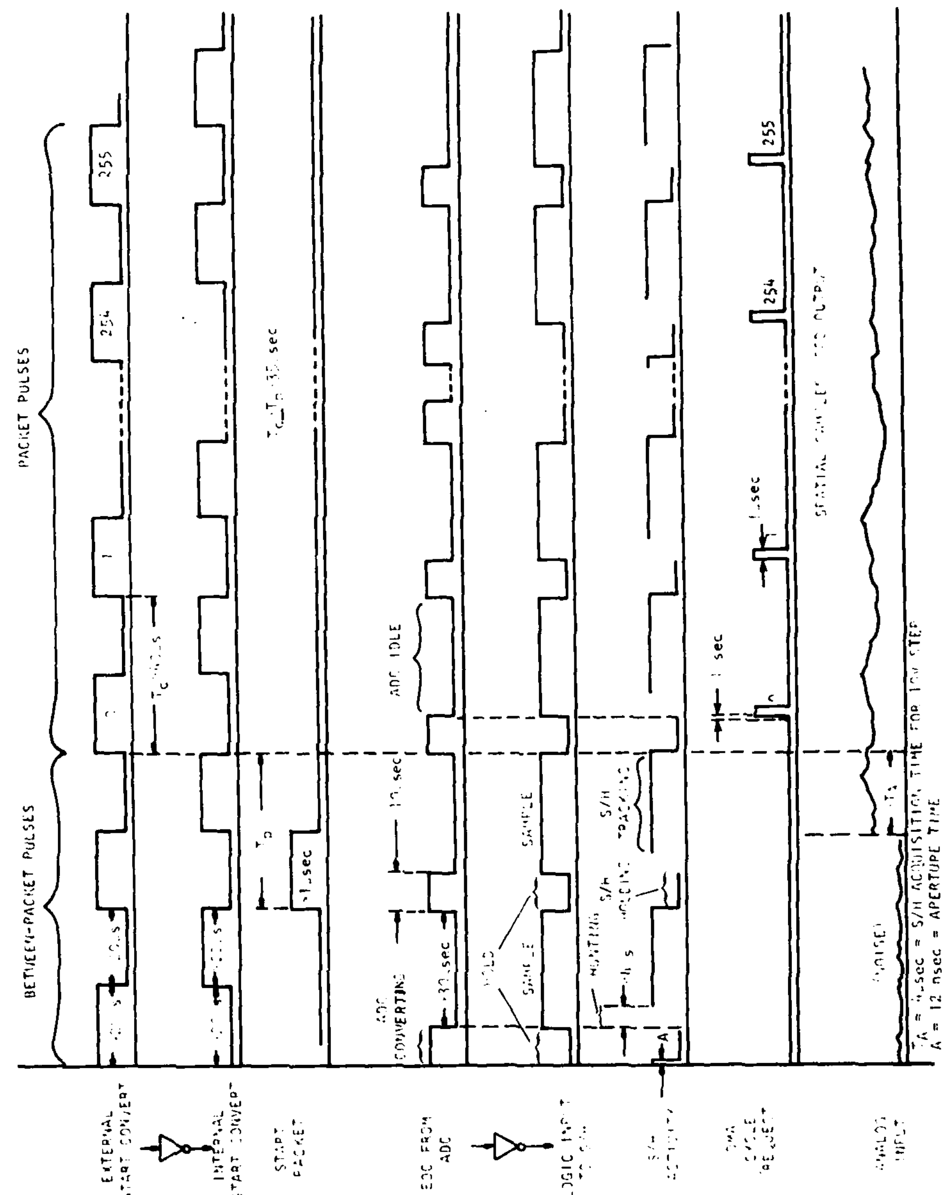


Figure 2.2 Timing diagram of IR-CCD interface operation.

The rising edge of start packet clocks the input "D" through to the output "Q" of the tandem connections of D-type flip-flops of the SN 74174. The inputs to these flip-flips are "function 1" and "function 2" supplied by the DMA board. Functions 1 and 2 are used to represent the appropriate stare time as determined by the CPU. The IR-CCD device receives a once-delayed version of functions 1 and 2 denoted as function 1(Δ) and function 2(Δ). The three AMD 25S10 TTL 4-bit shifter chips receive a twice-delayed version denoted function 1(ΔΔ) and function 2(ΔΔ). The purpose of the delays can be understood by looking at Figure 3.3 in Section III. The three AMD shifter chips are now properly aligned to shift the converted CCD voltages to the appropriate places. See table on wiring diagram, Figure 4.2.

The rising edge of the positive-going start packet pulse also triggers a SN 74123 one-shot producing a negative-going 18 μsec. pulse. The rising edge of this negative-going pulse triggers another one-shot producing a 1 μsec. positive-going pulse. The rising edge of this 1 μsec. positive-going pulse in turn triggers yet another one-shot (a high valued function 3 tied to its clear) producing a ≈ 142 μsec. positive-going pulse. The result of all this is the production of a ≈ 142 μsec. positive-going pulse that is delayed 18 μsec. after the rising edge of the start packet pulse. This long, positive-going pulse is used as the "clear" input to what we will call the "DMA one-shot." Only when the clear pin on any given one-shot is held high can the one-shot produce a pulse. The input to the DMA one-shot will be discussed shortly.

We turn now to the other digital signal coming from the IR-CCD device, namely the external start convert signal. This external start convert is once inverted using a Schmidt - Trigger inverter to produce the "internal" start convert signal which is sent to the ADC85C-12 converter. The falling edge of this internal start convert causes the "end of convert" (EOC) signal of the ADC85C-12 to move from the "ready" (low) state to the "busy" (high) state. Therefore, the falling edge of the internal start convert produces a positive-going ≈ 10 μsec. pulse for EOC. Ten μsec. corresponds to the specified time required by the ADC85C-12 to do a conversion to binary representation of an analog voltage. Eight μsec. is the observed time. The falling edge of EOC indicates the conversion is complete. An inverted version (again using a Schmidt-Trigger inverter) of EOC is used to change the SHM60 from the sample mode to the hold mode. A high to low transition causes this to occur. Therefore, the rising edge of EOC is used to <u>hold</u> an analog voltage, and the falling edge to signify end of conversion.

The falling edge of EOC triggers a one-shot producing a 1 μsec. negative-going pulse. The rising edge of this negative-going pulse triggers another one-shot producing a 1 μsec. positive-going pulse. It is this positive-going 1 μsec. pulse that acts as the input to the clearable DMA one-shot. The rising edge of this 1 μsec. positive-going pulse triggers the DMA one-shot (only if clear is high) producing positive-going 1 μsec. pulses that are used as "DMA Cycle Requests". When the DMA receives a Cycle Request, it loads a 16 bit word (16-bit CCD sample) into memory. The 12 ADC output data bits have been inverted (SN7404's) and sign replicated to form this CCD sample.

The software counts up to 255 (from zero) and then drops function 3 low <u>momentarily</u>. Dropping function 3 low will cause the ≈ 142 μsec. positive-going pulse to terminate prematurely. Since this long pulse was acting as the clear input to the DMA one-shot, the DMA one-shot will become disabled (i.e. no more Cycle Requests) as soon as function 3 goes low. Function 3 is immediately returned to its high state so as to be ready when the next start packet arrives. The high to low transition of function 3 is <u>not critical since DMA Cycle Requests after sample #255 are ignored by the DMA until after function 3 is momentarily set low</u>. At this point the interface board is ready for the next start packet to arrive.

In summary, start packet sets up the 4-bit shift chips by clocking the flip-flops and then turns on (starting with the first "good" CCD value) DMA Cycle Request pulses whose observed occurrence corresponds to ≈ 9 μsec. after each internal start convert falling edge (≈ 8 μsec. for conversion plus 1 μsec. delay). Function 3 turns off DMA Cycle Request soon after CCD sample #255 and reinitiates the IR-CCD interface board for the next start packet.

C. Signal Pin Assignments

Figure 2.3 indicates which signal appears at which pin of the 40-pin "berg" connector on the IR-CCD interface board. "Low" implies the pin is tied to ground and "high" to +5 volts. The wiring diagram of Section IV shows the location of this connector.

Figure 2.4 indicates which signal appears at which pin of the dual 40-pin berg connectors on the DMA board. The berg connector closest to the board end is J1. Figures 2.3 and 2.4 can be used to infer connections between the 40-pin IR-CCD berg connector and the two 40-pin DMA berg connectors.

Figure 2.5 indicates the pin assignment at the IR-CCD converter for ground, start packet, external start convert, function 1, and function 2 connections between the IR-CCD device and the interface board. This is a drawing (top view) of a 16-pin dip socket. Pins #1, #15, and #16 are the only common ground between the IR-CCD device and interface board. However, all other unused pins are also grounded at the interface board to provide a measure of noise shielding.

The analog voltages are supplied over an RG/174U coaxial cable. At this writing the shield of the cable is ungrounded at the IR-CCD device but is grounded at the interface board.

IR-CCD Micro-processor Interface Board

40-Pin Berg Pin Assignment

A	OPEN	Y	OPEN
B	DMA CYCLE REQUEST	Z	OPEN
C	OPEN	AA	OPEN
D	ATTN (LOW)	BB	OPEN
E	OPEN	CC	DATA BIT 7
F	A00 (HIGH)	DD	DATA BIT 8
H	WC INC ENB (HIGH)	EE	DATA BIT 6
J	BA INC ENB (HIGH)	FF	DATA BIT 9
K	FNCT 3	HH	DATA BIT 5
L	FNCT 3	JJ	DATA BIT 10
M	SINGLE CYCLE (HIGH)	KK	DATA BIT 4
N	C0 (LOW)	LL	DATA BIT 11
P	OPEN	MM	DATA BIT 3
R	FNCT 2	NN	DATA BIT 12 (LOW)
S	OPEN	PP	DATA BIT 2
T	C1 (HIGH)	RR	DATA BIT 13 (LOW)
U	OPEN	SS	DATA BIT 1
V	FNCT 1	TT	DATA BIT 14 (LOW)
W	OPEN	UU	DATA BIT 0
X	OPEN	VV	DATA BIT 15 (LOW)

Figure 2.3 Pin assignment of 40-pin "berg" connector on IR-CCD interface board.

DRV11-B DMA Interface Board

40-Pin Berg Pin Assignment

J1 INPUT CONNECTOR		J2 OUTPUT CONNECTOR	
A OPEN	Y OPEN	A OPEN	Y OPEN
B CYCLE REQUEST	Z OPEN	B OPEN	Z OPEN
C OPEN	AA OPEN	C OPEN	AA OPEN
D OPEN	BB OPEN	D ATTN	BB OPEN
E OPEN	CC OPEN	E OPEN	CC DATA BIT 7
F OPEN	DD OPEN	F A00	DD DATA BIT 8
H OPEN	EE OPEN	H OPEN	EE DATA BIT 6
J WC INC ENB	FF OPEN	J BA INC ENB	FF DATA BIT 9
K SINGLE CYCLE	HH OPEN	K FNCT 3	HH DATA BIT 5
L OPEN	JJ OPEN	L FNCT 3	JJ DATA BIT 10
M OPEN	KK OPEN	M OPEN	KK DATA BIT 4
N OPEN	LL OPEN	N C0	LL DATA BIT 11
P OPEN	MM OPEN	P OPEN	MM DATA BIT 3
R OPEN	NN OPEN	R FNCT 2	NN DATA BIT 12
S OPEN	PP OPEN	S OPEN	PP DATA BIT 2
T OPEN	RR OPEN	T C1	RR DATA BIT 13
U OPEN	SS OPEN	U OPEN	SS DATA BIT 1
V OPEN	TT OPEN	V FNCT 1	TT DATA BIT 14
W OPEN	UU OPEN	W OPEN	UU DATA BIT 0
X OPEN	VV OPEN	X OPEN	VV DATA BIT 15

Figure 2.4 Pin assignments of 40-pin "berg" connectors on DMA board.

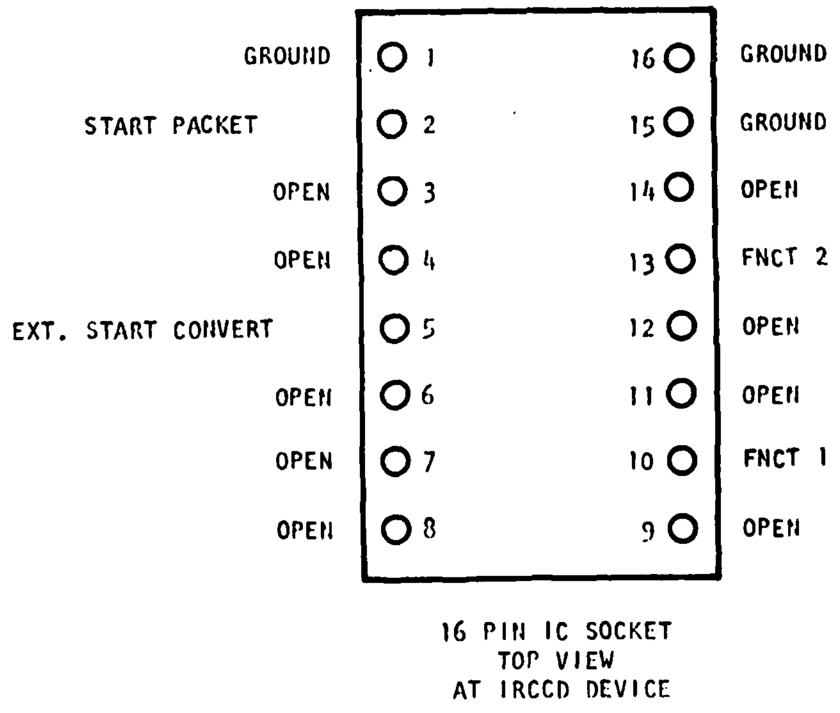


Figure 2.5 Pin assignment of 16-pin dip connector to IR-CCD device.

III. Software

The discussion of the IR-CCD Intrusion Detection System software will be divided into three main sections. The first of these will present an overview of the basic program philosophy and include a general discussion on the Input/Output procedures. The next section will concentrate on the actual implementation. This will contain a very detailed look at the computational routine, the heart of the system. Finally, the last section will undertake such topics as memory layout, partitioning of memory between MOS-RAM and PROM, etc.

A. Overview of Software Operations

A flow diagram for the main executive routine of the IR-CCD Intrusion Detection System is shown in Figure 3.1. The DMA "interrupt service" routine flow diagram is shown in Figure 3.2. These two diagrams illustrate the basic philosophy of the software by pointing out how the terminal handling function and computational function of the software interact.

Input/Output to the terminal is all software driven (versus interrupt driven). This means that the commands sent to the routine by the user and the messages sent by the routine to the user are carried on only when nothing else is going on. Thus, the computations are always given highest priority as these are interrupt driven by the DMA board.

When the main executive routine is not being interrupted by the calculations, it is in a continuous loop that is always checking (i) if there are characters being typed in by the user or (ii) if there are any characters that need to be sent to the terminal. If it is determined that the user has typed in a character, the subroutine "ttyin" is called. This routine stores the characters in a buffer, places them in an output queue to eventually echo back to the terminal, and, at the end of a command line, decodes the string of characters in the buffer and takes the appropriate action dictated by this string of characters.

If it is determined that there are characters which need to be sent to the terminal, the subroutine "send" is called. The "send" routine takes characters (or strings of characters) that have been placed on the output queue and in turn sends them to the terminal via the serial interface.

When an interrupt is received from the DMA indicating that a new packet of 256 data samples has been loaded into memory, program control is transfered from the main executive routine (or some subroutine that was ul-

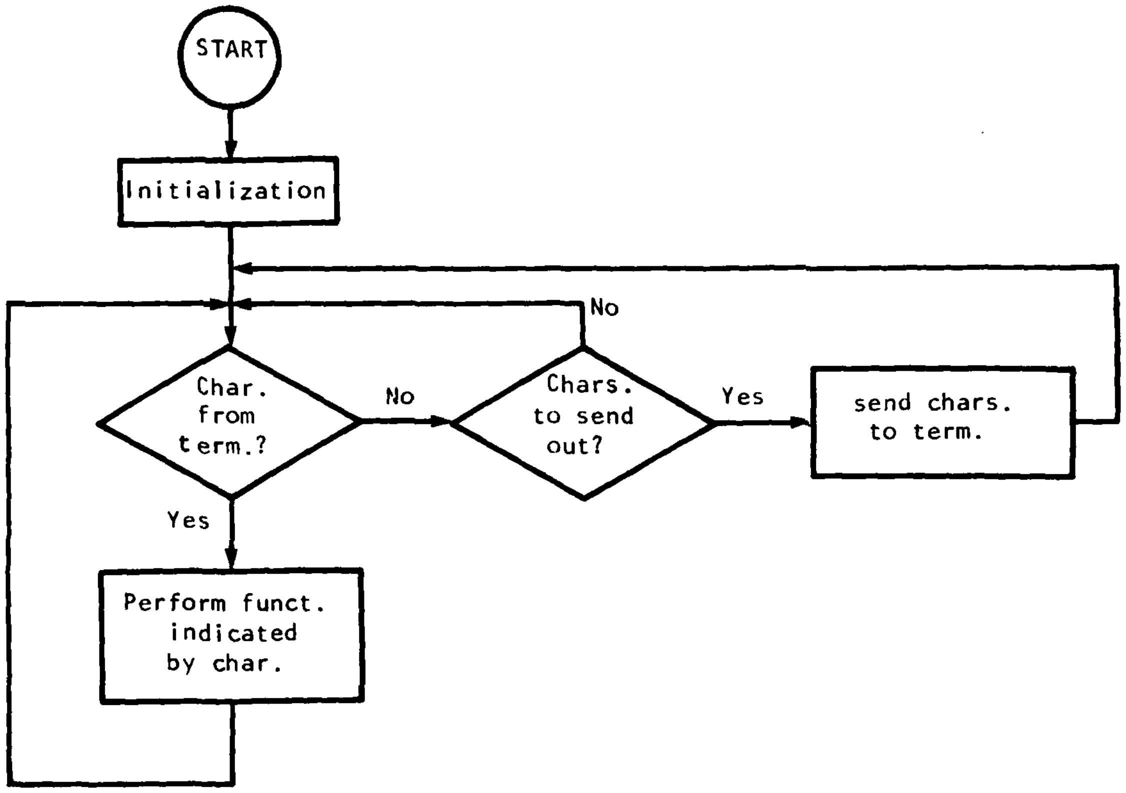


Figure 3.1 Flow diagram of main executive routine.

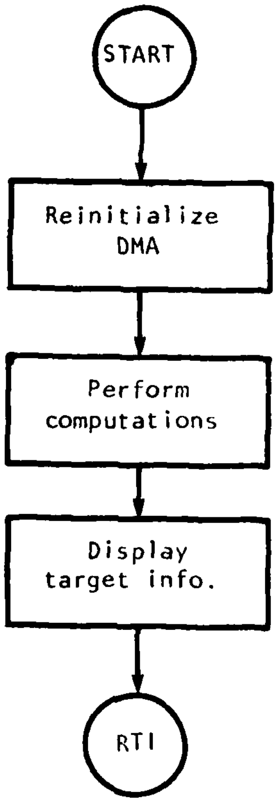


Figure 3.2 Flow diagram of DMA "interrupt service" routine.

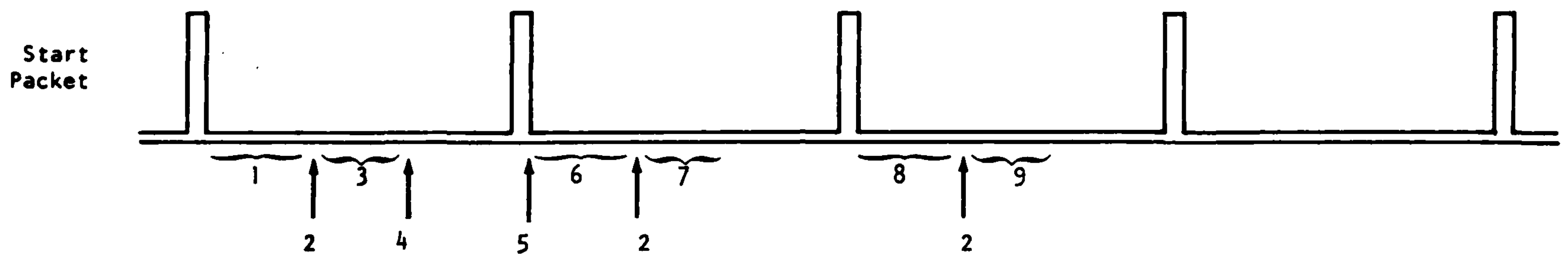
timately called from there) to the DMA interrupt service routine. This routine is called "endpk" in the assembly language program section. Upon entering the interrupt service routine, the first thing that must be done is to inhibit the cycle request pulses being received by the DMA from the IR-CCD interface. This is accomplished by toggling the function 3 line (controlled by bit 3 of the DMA Control Status Register) from high to low and then back again. This signal on the function 3 line is received by the hardware on the IR-CCD interface and the cycle request pulses are halted.

The DMA must now be reinitialized so that it is ready to start accepting the next packet of 256 data samples. This initialization is accomplished in several steps. First of all, the DMA Word Count Register (WCR) must be set to -256. This allows the DMA to perform 256 data transfers into memory for the next packet. Secondly, the "go" bit (bit 0) of the CSR must be set. This lets one data transfer occur every time a cycle request pulse is received by the DMA until all 256 transfers specified by the WCR have taken place for the next packet. Finally, the Bus Address Register (BAR) must be set to point to the location in memory where it is desired to start loading the next data packet. There are two buffers that are used to hold incoming data. These are called "ybuf1" and "ybuf2". By alternating between two buffers, it is possible to start loading in a new packet of 256 data samples before the computations on the previous packet have ended. The only requirement is that the computations on the old packet be completed before the computations on the new packet are begun.

Once the DMA has been reinitialized, the computational routine called "crunch" is called. This is an assembly language routine which actually implements the target detection algorithm of [2]. The "crunch" routine sets a flag which indicates whether or not a target has been detected, and then this flag is used by the "C" routine "irout" to display the proper target message on the screen. At this point an "RTI" (return from interrupt) instruction is executed and program control is transfered back to the main executive routine (or subroutine called from there) and terminal handling functions resume at the point they were before the interrupt occurred.

When the next 256 cycle request pulses have been received by the DMA, another interrupt request is generated, and the procedure is repeated. This continues to occur every time a new packet of 256 data samples has been loaded by the DMA into memory.

Another function that the DMA handles besides providing for very fast transfers of data into memory is that of allowing signals to be sent from the 11/03 to the IR-CCD electronics indicating a stare time change is needed. This is accomplished by using the function 1 and function 2 lines of the DMA (bits 1 and 2 of the CSR). Figure 3.3 illustrates the timing of this stare time changing sequence. When the computations calculate the background spatial average and it is determined that a stare time change is in order (by comparing this average with the thresholds C1 and C2), the software immediately sends out new values of the function 1 and 2 lines, corresponding to this new stare time, to the DMA board from whence they are sent to the IR-CCD interface. At the start of the next packet, the hardware passes these new function lines to the IR-CCD electronics. Since the old stare time was still in effect when this new packet was optically integrated by the IR-CCD array, it is not necessary to check again for a stare time change. When the <u>second</u> packet <u>after</u> the one where a stare time change was deemed necessary is ready to be converted and loaded into memory, the new function line values are sent by the hardware to the programmable digital shift logic since this second packet is the first one to be optically in-



1. Packet #n being loaded by DMA into memory.
2. DMA interrupt hits.
3. Computations decide stare time needs changing.
4. Software sends out new function 1 and function 2 values.
5. IRCCD receives new function 1 and function 2 values and thus the accumulation of packet #n + 2 is lengthened or shortened accordingly.
6. Packet #n + 1 being loaded by DMA into memory.
7. Software does not check for stare time change.
8. Packet #n + 2 being loaded by DMA into memory. This is the first packet which uses the new stare time and hence is shifted accordingly by the interface electronics.
9. Computations check for stare time change using new C1 and C2 values.

Figure 3.3 Timing of store time changing sequence.

tegrated under the new stare time. The background spatial average is again computed using this new packet loaded in under this new stare time and this value is compared to scaled (multiplied or divided by 2) values of the previous C1 and C2 threshold values.

This completes the overview of the software operation. The next section will cover the actual code that implements the terminal handling and computational routines. For more detailed information on use of the DMA, the reader is referred to [3].

B. Implementation of Software

This section explains how the various functions of the IR-CCD Intrusion Detection System were implemented in software. The coding of the terminal handling routine will be covered first, followed by a discussion of the assembly language computational routine. Some more detailed remarks regarding these routines may be found by examining the comments in the listings given in Appendix B.

The main executive routine of the terminal handling section of code was covered in some detail in the previous section so this discussion will not be repeated here. The remaining routines will be discussed one at a time in the approximate order in which they might be called during typical use of the IR-CCD Intrusion Detection System.

(1) initial - This routine performs all initialization for the IR-CCD Intrusion Detection System. It is called at turn on time and when the computations are restarted after performing one of the self-diagnostic tests. The first thing that is done is to disable the interrupts from the DMA. This prevents the calculations from being entered before everything is ready. The DMA is initialized at this time following the same procedure as discussed previously in the overview of software operation. Pointers to various input buffers and output queues are set as well as the initial values for all of the flags that are used to keep track of the status of various events. The Tektronix 4024 terminal is then initialized. This involves dividing the screen into appropriately sized workspace and monitor sections, programming the "PT" key to issue the "20000G" sequence, and programming the various delete character and delete line keys. At this point it is necessary to see if the DMA maintenance cable is in place. This is

done by sending out a bit pattern on the three DMA function lines and checking to see if it is echoed back by the three DMA status lines (as would be the case if the cable were in place). If the transmitted and received bit patterns are the same, it is assumed the maintenance cable is in place and the DMA test is entered by calling the routine "dmatst". If the two bit patterns are not the same, the rest of the initialization is performed.

The initial parameter display frame is now constructed on the screen by calling the "display1", "bss", "puttty1", "box", "alarm", and "send" routines. After this, all initial parameter values are set by storing their binary value in "datareg" and the ascii representation of this value in the character array "datstrng". This allows the parameter value to be displayed on the screen when the appropriate parameter changing routine (ddsrv, nnsrv, ppsrv, dsrv, nsrv, tsrv, C1srv or C2srv) is called. The convention assumed for all ascii strings is that they will be terminated in a 0.

The final step in the initialization procedure involves the DMA. When DMA interrupts are enabled, an interrupt is immediately generated due to the DMA being in the ready state. Therefore the first interrupt that is received causes program control to go to a dummy interrupt service routine (appropriately called "dummy") which sets the interrupt vector pointing to the actual end of packet interrupt service routine. This dummy interrupt service routine also sends out the initial values of the DMA function 1 and function 2 lines corresponding to the stare time "tref". After this, the alarm disable flag is set to allow the background time average to be computed before enabling the alarm, the "go" bit of the DMA is set so that the DMA is now ready to accept data samples, and the function 3 line is sent high to inform the IR-CCD custom interface that the software is now ready.

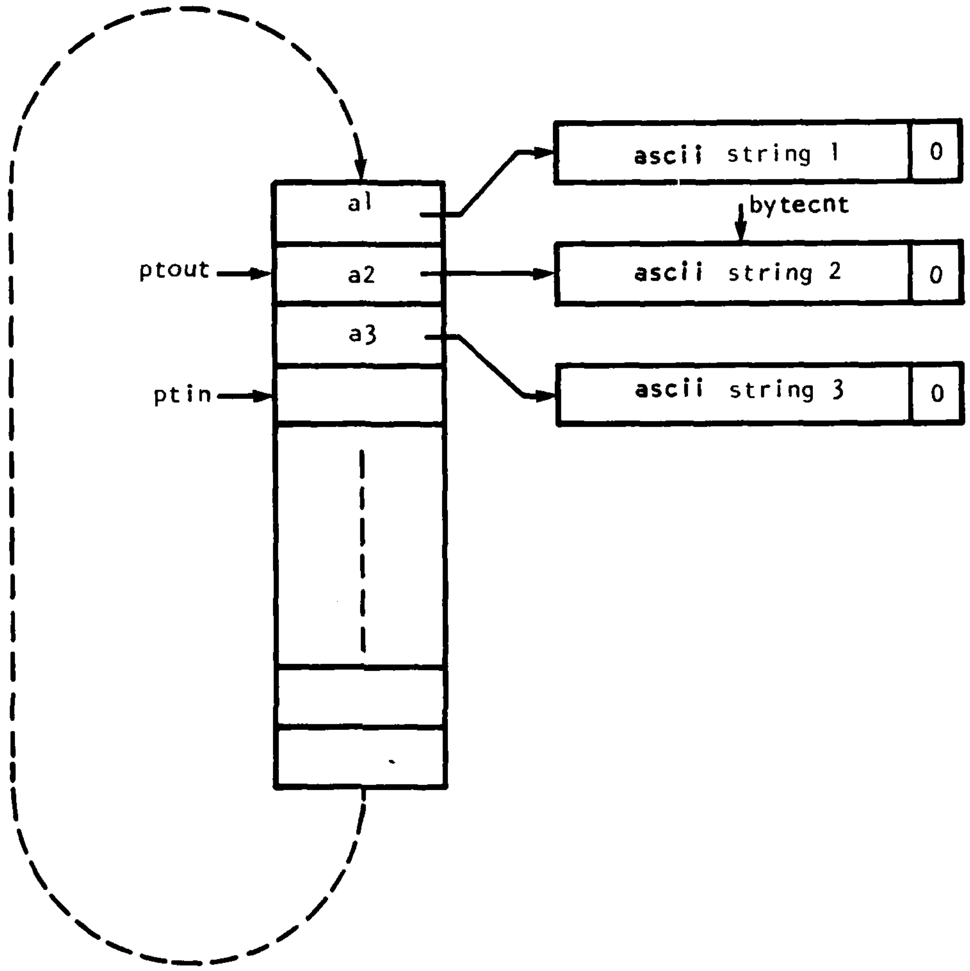
(2) ttyin - This is the routine that is primarily responsible for accepting

character data from the terminal. Called from the main executive routine whenever a character is present at the input of the serial interface, this routine first places characters into a temporary buffer until an end of command character is received (carriage return). As each character is received, it is tested to determine if it is a special erasure character (delete line or delete character) and if so, the appropriate action is taken. If the incoming character is not a special one (erasure or carriage return), the character is echoed back to the terminal by calling the "puttty1" subroutine. The actual sending of the character is handled by the main executive routine via the calling of the "send" subroutine. When a carriage return is detected, the subroutines "parse" and "decode" are called. These routines interpret the meaning of the incoming character string and return a number that corresponds to that string. The value of this number from the previous command string (stored in "oldcomnd") is used for detecting an invalid sequence of parameter or mnemonic entries. When an error of this type is detected, an appropriate error message is sent to the screen, again via the "puttty1" subroutine. At this point, two C "switch" statements are used to go to the proper subroutine for handling the operation requested by the incoming command string. If the current string is decoded and found to be a valid parameter mnemonic or carriage return, a routine is called which lists the possible parameter values in the case of receiving a valid parameter mnemonic, or, in the case of receiving a carriage return, the routine "addfeat" is called to display the additional features frame. If a numeric string is received, the routine "decode" calls another routine "ascton" which converts the incoming ascii into a binary number (appropriately formatted according to the previously typed command). The second "switch" statement is then used to enter the proper parameter changing routine or, if expecting an additional feature selection, the routine "lsrv" which in turn goes to the selected additional feature executing routine. If an error of any kind is detected at any step along the way (such as the routine receiving two numeric character strings in a row) an error message is sent to the screen via the "puttty1" routine.

(3) puttty1 - This is a routine which puts the starting address of a zero terminated ascii string onto a circular queue. All messages to the terminal from user controlled functions are handled through this routine. The pointer "bfpti1" is used to point to the location in the queue where the next starting address of a string is to be placed. The operation of this queue in handling output to the terminal is illustrated in Figure 3.4.

(4) puttty2 - This routine is identical to "puttty1" except that it is used to handle messages to the terminal from interrupt controlled routines. This primarily involves "no target/target detected" messages. The use of a separate output queue for messages of this type prevents the interruption of the displaying of normal ascii strings (from user controlled functions) by the arbitrary occurrence of an interrupt controlled message display operation.

(5) send - This routine performs the function of emptying the two output queues and actually sending the ascii strings to the terminal. Characters are accessed by finding the starting address of the ascii string being sent out. This pointer to the string is in turn pointed to by "ptout". See Figure 3.4 again. A counter called "bytecnt" is used to access the next character in the current output string. When the end of line character (0) is reached, "ptout" is moved to point to the next location in the circular queue. If this location is the same as the location that the input pointer



'ptin' points to location where next address of an ascii string will be placed.
'a1' points to first character in zero terminated ascii string 1.
'a2' points to first character in zero terminated ascii string 2.
'a3' points to first character in zero terminated ascii string 3.
'ptout' points to location where pointer to next ascii string to be sent to terminal resides.
'bytecnt' points to the next character in the string that is currently being sent out.

Figure 3.4 Structure of output queues.

"ptin" is pointing to, then all messages have been sent out and the output queue is empty. When both output queues have been emptied, the "send" routine ends. The routine is somewhat intelligent in that it does not just arbitrarily send characters to the terminal. When a 4024 "!Jum" command is detected as being sent from the interrupt controlled output queue, it is necessary to follow this command with several ascii nulls to allow the terminal to catch up during rapid message changing conditions. In data sent to the output queue from user controlled functions, there are several often repeated ascii strings which are replaced by special single ascii characters to save storage space in PROM. These special characters and the ascii string with which the "send" routine replaces them are listed in Table 3.1. The lines of repeated underscore characters in the parameter display list are also coded in a special form which the send routine decodes. The convention used is that if an underscore is detected, the next character will specify how many times the underscore should be repeated.

(6) parse - This routine is used to ensure that the various typed in command lines are all in the same format before attempting to decode these strings in the "decode" subroutine. There are two primary functions that this routine handles. When a carriage return is typed, it may mean one of two things: (i) a simple end of command line or parameter entry terminator, or (ii) an actual command such as when toggling between the various screen frames. For a numeric string, a carriage return would be interpreted by the decoding routine as a non-numeric character and hence generate an error. Thus, command and parameter type strings are placed in a buffer with the carriage return deleted and an end of line indicator (0) placed after the last valid character. When a carriage return is a valid command, it is placed in the buffer followed by the terminating 0. The second function of

Ascii code	Function or Ascii String that Replaces Special Character
001	"!JUM"
002	"No error detected in"
003	"Error detected in"
004	cursor positioning information
005	"Hit 'RETURN'"

Table 3.1 Special ascii characters and their corresponding messages.

the "parse" routine is to remove any leading blanks that may have been typed in before placing the string in the buffer.

(7) decode - The command decoding routine serves several different functions. The string of ascii characters that was placed in a buffer (called "string") by the "parse" routine is the primary input to this routine. If this string is a valid command (such as a parameter mnemonic, single carriage return, etc.), the string is interpreted and a numeric value is returned that corresponds uniquely to that command. If the string if found to be numeric in nature (such as when a parameter value is typed or when an additional feature is selected), the routine "ascton" is called. This routine converts the ascii representations of numeric values into appropriately formatted binary representations. If an error was found in converting the number from ascii to binary (such as more than one decimal point, a non-numeric character, etc.) an error flag is returned by "ascton" and in turn passes the error indicator on to the "ttyin" routine by returning the value 0 as the command number. If the conversion to binary is successful, the value 10 is returned. The value 0 is also returned if an invalid command mnemonic is detected. Returning these different values back to the "ttyin" routine informs that routine whether or not one of the keyboard entry servicing routines (ddlist, nnlist, ddsrv, lsrv, etc.) should be called, and also which routine is appropriate.

(8) ascton - This routine converts ascii strings into appropriately formatted binary numbers. All numeric data entry into the IR-CCD Intrusion Detection System from the keyboard is in integer values except for changes to the "d" parameter. The flag "oldcomnd" allows this routine to know which kind of data to expect by storing the number returned by the "decode" call of the

most recent valid command line. The actual typed entry for any numeric value may be either integer or floating point. If expecting an integer value, the "ascton" routine rounds it off to the nearest integer. Similarly, when expecting a new value for the "d" parameter, "ascton" rounds the entry off to 2 decimal (7 binary) places. Thus the proper format for the "d" parameter as stored in binary is bbbbbbbb.bbbbbbb. There are two locations in which the rounded off value of the data entry is placed. One is in a buffer "datareg" which holds the formated binary representation and the other is a character array buffer "datstring". The ascii characters stored in "datstring" may be different than those that were typed in. This is due to the rounding operation described previously. All of the routines which expect numeric data look for it in the "datareg" buffer and assume that the binary is already in the proper format needed for that routine. Also, the ascii representation stored in "datstring" is transfered to another buffer by these same parameter servicing routines. This allows the screen to be refreshed without having to convert the binary representation back to ascii. A much more detailed description of the inner workings of the ascton routine is contained in the program comments.

(9-16) ddlist, nnlist, pplist, dlist, nlist, tlist, C2list, C1list - All of these routines are identical in nature and therefore will be discussed together. When a valid parameter mnemonic is typed, a list of possible parameter values must be displayed at the bottom of the screen. These various "list" routines perform this function. In each subroutine is a call to the "bss" routine. This "bss" routine blanks the bottom of the screen (4024 monitor space) and refreshes the instruction line telling how to display the list of additional features. The "oldcomnd" flag mentioned previously in the "ttyin" and "ascton" sections is set in these listing routines.

(17-24) ddsrv, nnsrv, ppsrv, dsrv, nsrv, tsrv, C1srv, C2srv - These are the routines that are entered from the "ttyin" routine actually to perform the parameter changing operation. When each routine is entered, the new parameter value is stored in the buffer "datareg" from the "ascton" routine. This value is then tested to determine if it is one of the allowed parameter values for the variable that is going to be changed. If not, an error message is sent to the screen via "puttty1" and the user is again prompted with the list of possible parameter values. When it is determined that the typed entry is a valid parameter, several things can occur. With some parameters there are auxiliary quantities that must be computed for use by the computational routine. If the computational routine is entered before all of these auxiliary quantities have been changed (which can happen due to these computations being entered under interrupt control), confusion will result. Therefore, on some parameter changes (D, N, P, n, C1 and C2) the computational routine is temporarily disabled by use of the flag "intflg". The new parameter value may now be moved from the temporary buffer "datareg" and placed in a permanent storage location. The auxiliary quantities are also computed for those parameters where it is appropriate. These will be described in more detail in the section describing the computational routine. When either N or P is changed, the computations must be started from the beginning due to the need for a new background time average. This means that arrays that store information dependent on or used in computing the background time average must be reset to their starting state and in some cases cleared to zero. More will be said about these arrays ("x", "m", and "a") in the section describing the computational routine. In addition to transferring the parameter value from a temporary buffer, the ascii string representing this number which was placed in "datstrng" in the "ascton" routine is transfered to a permanent location. Storing this string allows the parameter values to be displayed at any time without having to convert back from binary to ascii. When all of the necessary steps have been completed in changing a parameter (completed as far as the computations are concerned), the new parameter value is sent to the screen via "puttty1" and the prompt at the bottom is changed back to the instructions on how to initiate a parameter change

(25) adfeat - This is a routine which constructs the additional features frame on the terminal screen by sending a number of stored ascii strings to the terminal via "puttty1". Calls are made to the "box" and "alarm" routines to aid in this process. The subprogram "box" draws the alarm status box at the bottom center of the screen while "alarm" fills in the box with the appropriate message.

(26) putdsp1 - This routine constructs the parameter display frame on the terminal screen by sending a number of stored ascii strings to the terminal via "puttty1". Calls are made to the "display1", "bss", "box," and "alarm" routines to aid in this process. The routine "display1" draws the parameter table at the top of the screen, "bss" fills in the prompt messages at the bottom, "box" draws the alarm status box, and "alarm" fills in the box with the appropriate message.

(27) display1 - This routine aids "putdsp1" by constructing the parameter display table at the top of the parameter display frame.

(28) box - This routine constructs the alarm status box at the bottom center of the various frames displayed on the terminal.

(29) alarm - This routine examines the flag "almflg" indicating the current

status of the alarm (disabled, no target detected, or target detected) and then displays this message inside of the alarm status box constructed by the "box" routine.

(30) bss - This routine clears the bottom of the screen (monitor) and sends the "Type L to see list of additional features" line to the terminal via "puttty1".

(31) bds - This routine clears the bottom of the screen (monitor) and sends the invalid parameter error message to the terminal via "puttty1".

(32) help - This routine is called from the "end of packet" interrupt service routine "endpk" when it is determined that the calculations are falling behind. The purpose of this routine is to inform the user of this condition by displaying an appropriate message on the terminal via "puttty2".

(33) irout - This routine is called at the end of the computations and determines if the message presently in the alarm status box is different from the one called for by the just-completed computations. This allows the message in the alarm status box to be updated only if there is a change needed thus preventing the terminal from getting bogged down. "irout" also sends out the audible alarm (when enabled) for every packet for which a target is detected. This sustains the audible alarm for the entire time that a target is present.

(34) lsrv - This routine is used to service the additional features list. If feature 1 is selected (the toggling of the audible alarm status), an appropriate flag is set and "lsrv" is exited. For the remaining features, the desired test routine is entered <u>except</u> for the DMA test. In that case only the DMA test instructions are displayed and the 11/03 "halt" instruction is

issued if the user desires to perform the test.

(35-40) atodins, pdsins, termins, promins, ramins, dmains - These routines are used to display a brief description and/or instructions for each of the diagnostic test routines. There are also calls made to the "box" and "alarm" routines to allow the alarm status box to remain updated while the test descriptions are on the terminal screen.

(41) atodtst - This routine tests the A to D converter on the custom interface board by loading samples into memory via the DMA and then displaying these samples on the terminal screen. The two function lines are set to zero to allow all 12 bits of the A to D converter output to be displayed on the screen. The routine "atod" performs the actual data transfer and conversion to ascii, and is called repeatedly in a while loop until a character is received from the terminal. Once the test has been completed, the initialization routine "initial" is called to restart the program from the beginning.

(42) atod - This is the routine that actually controls the reading and display of samples from the A to D converter. This code is called by both the "atodtst" and "pdstst" routines. The first thing that is done in this routine is to set up the DMA for a single word transfer into memory. Once the "go" bit of the DMA is set, the routine waits for a start convert pulse to be received by the custom interface board. That initiates the actual data transfer. When the DMA has completed loading the word into memory, the conversion from binary into ascii takes place. The word is converted into the ascii representations of both its binary and decimal values. The decimal conversion is handled by the routine "atodec". These two representations are then displayed on the terminal screen via the "puttty1" and "send"

routines. A slight delay is added at the end to prevent the 4024 terminal from falling behind.

(43) pdstst - This routine tests the programmable digital shift (PDS) logic by repeatedly cycling through all possible values of the function 1 and 2 lines, loading these shifted samples into memory via the DMA, and finally displaying these values on the terminal screen in both binary and decimal. Initially both function lines are set to ones (corresponding to tref). New values are clocked in using the start packet signal. The actual data at the output of the PDS logic is loaded into memory via the "atod" routine where the results are then displayed on the screen after the data transfer takes place. The current values of the two function lines are then displayed on the screen before it is determined if it is time to update these lines to new values. The test continues until any character is received from the terminal via the serial interface. At this point, the initialization routine is called and the program is effectively restarted from the beginning.

(44) termtst - This routine tests the Tektronix 4024 terminal and the serial interface to which the terminal is connected. The terminal is tested by issuing the standard 4024 "test" command. For more information about this command see [4]. The next segment of this routine tests the serial interface and monitor section of the 4024 screen by repeatedly sending an 81 character long string of ascii data to the terminal. The "puttty1" and "send" routines are used in this process. There is a short delay added after sending each line to prevent the 4024 from lagging behind what is being sent out by the microprocessor. The last section of this routine tests the serial interface and workspace section of the 4024 screen using the same

method as was used to test the monitor section of the screen. As with most of the other test routines, the program is effectively restarted after completion of the test by calling the initialization routine "initial".

(45) promtst - This routine computes two 16 bit checksums, one for the lower 4k words of PROM and one for the upper 4k words of PROM. The expected values of these checksums are stored in two words in the upper 4k block of PROM. This section of PROM is not used in the computation of the checksum for the upper 4k block of memory. Once these two checksums are computed, they are converted to the ascii representation of their octal value by use of the "atooct" routine. In a similar fashion, the expected values of these checksums are retrieved from memory and converted to ascii by calling the "atooct" routine. In addition to displaying these actual and expected checksums on the terminal screen, comparison is made of these values by the microprocessor and an appropriate set of error/no error messages is displayed on the screen. Upon receiving any character from the terminal, the initialization routine is called to restart calculations and the "promtst" routine is exited.

(46) ramtst - This routine is used to test the integrity of each word in the main section of RAM by writing unique bit patterns into a location and then trying to read these patterns back to see if they were stored correctly. This testing procedure is accomplished using three of the general purpose registers to prevent destroying any useful information that had been previously stored in RAM (specifically the stack which is used to hold all subroutine calling information). One register is used to point to the desired test location, one register is used to hold the original bit pattern, and the third register acts as a temporary buffer to store the origi-

nal contents of the desired test location. If an error is detected during the test, the location of the error (the contents of the register that is used to point to the test location) is converted to the ascii representation of its octal value using "atooct" and this value is displayed on the screen. The routine then waits for either a continue symbol to proceed with the test or a terminate symbol to end the test. Whenever an error is detected, an accummulator is incremented to keep track of the total number of errors detected. When the test has been completed (either by having tested all RAM locations or having the test prematurely terminated by the user) this accummulated error count is converted to decimal by "atodec" and displayed on the screen. Finally, the initialization routine is called to restart the calculations and the "ramtst" routine is exited.

(47) dmatst - This routine is used to call repeatedly the routine "dmats" which in turn calls the actual DMA test routine which is written in assembly language. A call is made to "dmats" every time a character is received from the terminal. The "dmatst" routine is entered from the routine "initial" when it has been determined that the DMA maintenance cable is in place.

(48) dmats - This routine is used to handle the output end of the DMA test. The actual test routine is written in assembly language and is called "dma". In "dmats" there are seven global flags, one for each function of the DMA that is to be tested. These flags are originally set to zero. When the assembly language routine "dma" is called, these flags are either left at zero (if no error for the corresponding test is detected), or set to a one (if an error is detected). After control has been passed back to the "dmats" routine, these seven flags are examined and an appropriate error/no error message is displayed on the screen for each segment of the test. At this

point, control is returned to the "dmatst" routine.

(49) dma - This is an assembly language routine that actually performs the tests on the various DMA functions. The user is referred to [3] for more complete details about DMA operation and terminology. The first three tests that this routine performs involve verifying the read/write capabilities of the Word Count Register, the Bus Address Register and the Data Buffer Register. This is done by first writing into each of the various registers an alternating zero-one pattern. The WCR, BAR and DBR are then read back and the results compared to the original pattern. If differences are detected, the appropriate error flags are set and the "dma" routine continues with the next test. If no errors are detected, the zero-one pattern is complimented and the test repeated.

The next DMA function that is tested is the clearing of various registers when a bus INIT signal is sent. This is accomplished by setting all possible bits in the WCR, BAR, Control Status Register, and DBR and then issuing the "reset" command to send the bus INIT signal. The four DMA registers are then examined to see if they have been set to the proper values (these are not all zero since some bits always read as ones). If all four registers do not check out properly, an error flag is set.

The function-status lines are the next DMA section to be tested. When the DMA maintenance cable is in place, the three function lines are fed back into the three status lines and hence the status lines should echo back what is sent out on the function lines if all is working properly. This portion of the "dma" routine sends out all possible combinations of function line values and verifies whether or not the correct values are received by the three status lines. If all tests are not successful, an error flag is set.

The final two DMA tests involve the transferring of data to and from memory and the issuing of interrupts at the end of data transfers. These operations are again accomplished through the use of the DMA maintenance cable. When the maintenance, interrupt enable, and go bits of the DMA CSR are set, and the WCR and BAR have been properly loaded, the "BUSY" line feeds back into the "CYCLE REQUEST" line to initiate data transfers which alternately takes a word from a memory location pointed to by the BAR <u>or</u> places this word in the next consecutive memory location. This cycle repeats itself until the WCR has been incremented to zero. At that time, an interrupt request should be issued by the DMA and program control should transfer to the DMA interrupt service routine "dmaint". This assembly language routine sets a flag to acknowledge that the interrupt request has been honored and then returns control back to the "dma" routine. After a short delay to insure that the interrupt has had enough time to occur and to be serviced, this flag is examined and if it is found to have not been set by the "dmaint" routine, an appropriate error flag is set. The final part of the "dma" routine examines the locations in memory where data transfers should have occurred, and verifies whether or not the proper values have been loaded into the proper locations. An error flag is set when an unsuccessful transfer is detected. Program control is then returned to the "dmats" routine where the seven error flags are examined and appropriate messages displayed on the terminal screen.

(50) atodec - This routine takes a 16 bit word and converts it into a character string that contains the ascii representation of the decimal value of this word. Negative numbers are first changed to positive numbers before the actual conversion takes place. The conversion process itself involves repeatedly performing integer divisions by decreasing powers of ten. The

results of these divisions are the individual digits in the binary coded decimal representation of the original word. These digits are in turn converted to ascii by adding 60 (octal). As with all ascii strings, a zero is added to the end for a string terminator.

(51) atooct - This routine takes a 16 bit word and converts it into a character string that contains the ascii representation of the octal value of this word. The conversion is performed by first masking off groups of three bits to get the individual octal digits, and then converting these digits to ascii by adding 60 (octal). A trailing zero is added to the string as a terminator.

(52) thresh - This is an assembly language routine that computes a table of decision threshold values corresponding to background time averages in the range 0 to 1023. The basic method for computing the needed square roots is a Newton-Raphson type scheme. That is, the roots of the equation

$$f(X) = X^2 - a = 0$$

are found by performing ten iterations of the equation

$$X_{n+1} = X_n - \frac{f(X_n)}{f'(X_n)}$$

or

$$X_{n+1} = X_n - \frac{X_n^2 - a}{2X_n}$$

After these iterations, the value of X will be approximately equal to the square root of a. Throughout this routine, numbers are carried around with an appropriate binary format to ensure integer accuracy in the final threshold values.

Once the above square root is found, it must be multiplied by two other factors. These are (i) the square root of the quantity N+1/N and (ii) the d parameter. The first of these factors was found via table look up in the "nnrsv" routine and is stored in the variable "sqn". The other factor was stored in "smd" in the "dsrv" routine and has a format containing seven binary places. After these three terms are multiplied together using the fixed point multiply instruction, the resulting threshold is placed in the proper location in the threshold table "btab" pointed to by "bstr".

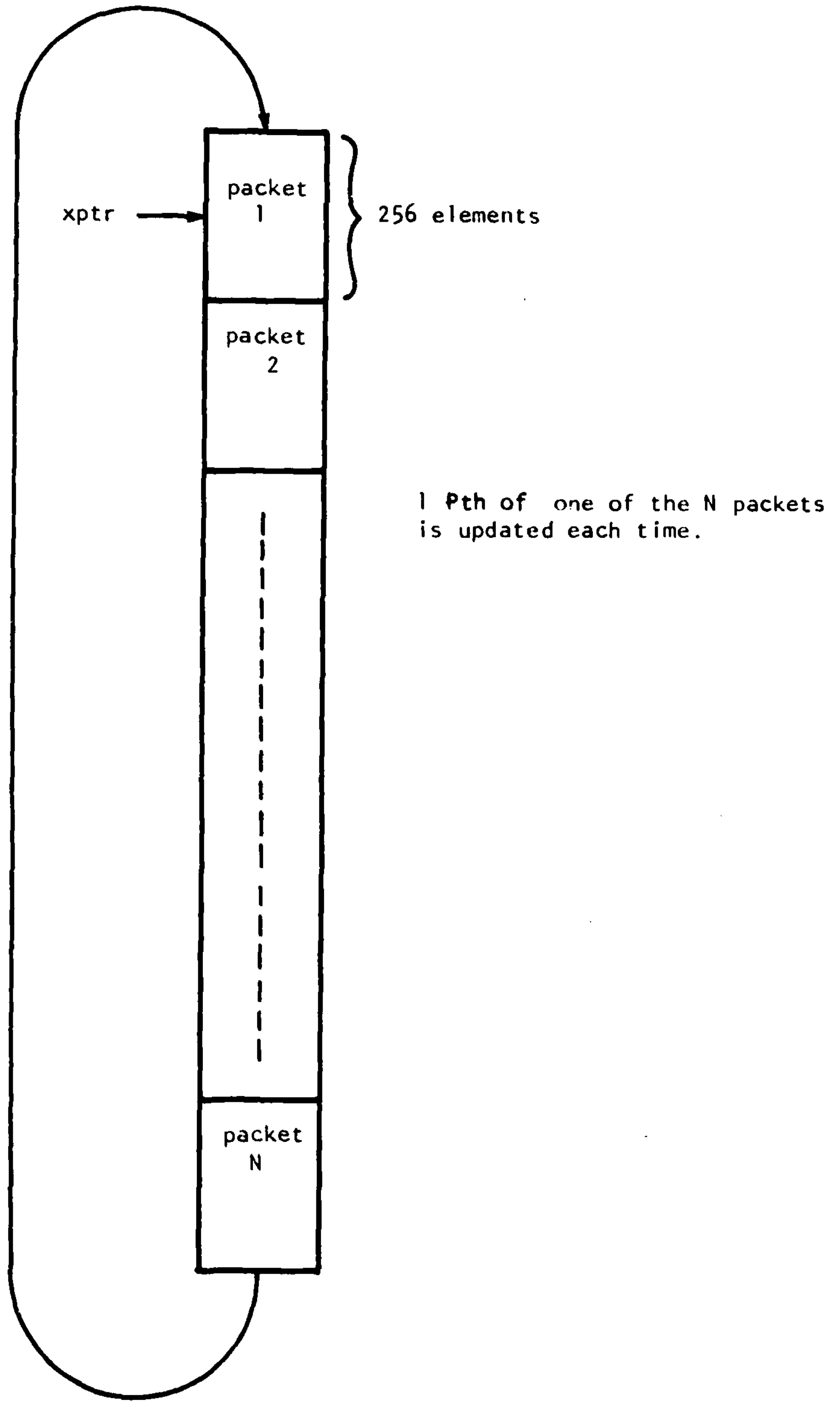
(53) dmaint - This is a DMA interrupt service routine that is entered during the DMA diagnostic test. The only function it serves is to set a flag indicating that the DMA interrupt service request has been honored.

(54) halt - This is an assembly language routine that merely executes the 11/03 "halt" instruction. This allows the "C" routine "lsrv" to halt the program when the user desires to perform the DMA diagnostic test.

This concludes the discussion of what have roughly been called the terminal handling routines. As mentioned previously, more detailed comments are contained in the program listings in Appendix B.

The actual implementation of the computations performed in the IR-CCD Intrusion Detection System will now be discussed. This will begin by describing three main vectors which are used to store various quantities used in the calculations. The first of these vectors is one called "x". This may be thought of as a first in-first out circular queue that stores the past data samples used in computing the background time average. Roughly 1 P$\underline{\text{th}}$ of these samples are updated each pass so that in P passes one sample from each spatial element of the IR-CCD array has been stored. There are N of these samples from each element that must be stored thus giving the queue a length of N*256 words. This queue may be thought of as being broken down into N packets with 256 elements in each packet and each packet being updated over the course of P passes. See Figure 3.5. There is one pointer, called "xptr", which points to both the location where the oldest element in the queue may be found and also where the newest incoming element is to be placed. This allows the background time average to be computed by subtracting the oldest sample from an accumulated sum of samples and then adding in the newest sample. This newest sample is then stored in the same location where the oldest sample was removed.

The second important vector is one called "m". This vector may be thought of as being divided into four distinct blocks of 256 words each. The first and last 256 word blocks are actually storing different parts of the same quantity. This quantity is the accumulated double precision sum of terms used in computing the background time average. In other words, this is the background time average before it is divided by N. The first 256 words of "m" store the low order word of the double precision sum while the last 256 words store the high order word of this sum. See Figure 3.6. Each word of a block corresponds to one cell of the IR-CCD array.



'xptr' points to the location where the oldest element in the queue is to be removed and where the newest element is to be placed.

Figure 3.5 Layout of "x" vector.

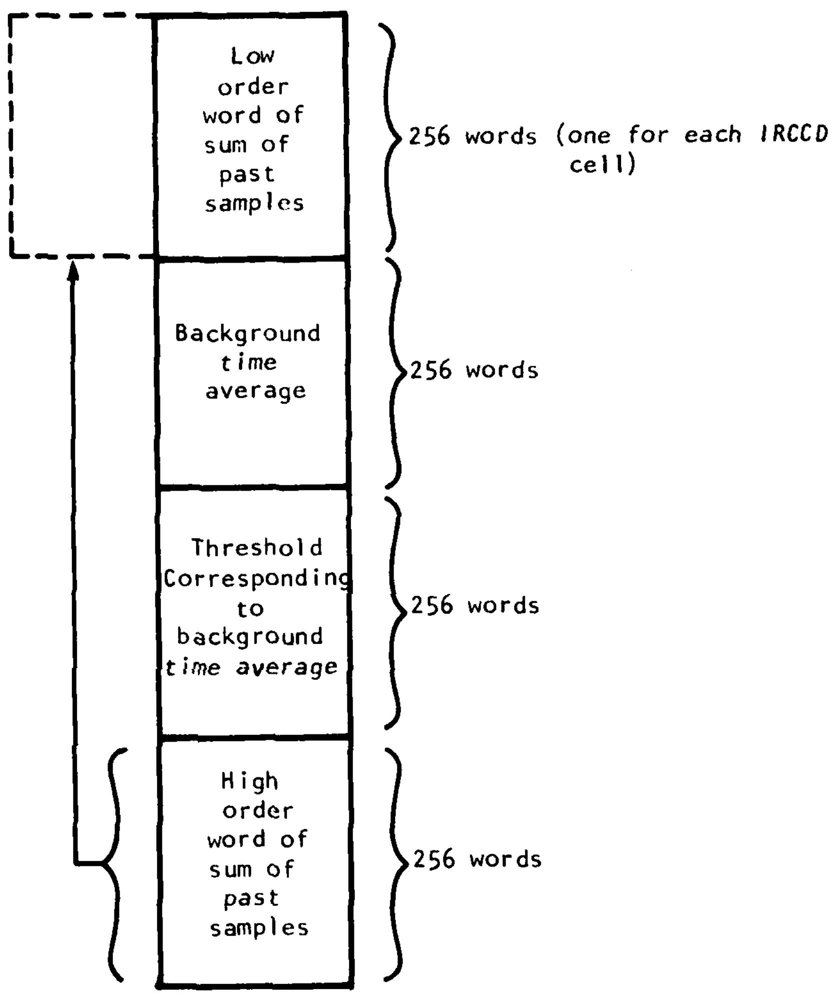

Low order word of sum of past samples
256 words (one for each IRCCD cell)
Background time average
256 words
Threshold Corresponding to background time average
256 words
High order word of sum of past samples
256 words

Figure 3.6 Layout of "m " vector.

The second 256 words of the "m" vector store the actual background time average. This is just the corresponding element of the first (and last) block of "m" divided by N.

The third 256 words of "m" store the decision thresholds for each of the background time averages in the second 256 word block. This decision threshold is found via a table look-up procedure.

The last important vector used in the calculations is the "a" vector. This 256 word vector stores the past history of threshold violation information. Each word corresponds to a cell of the IR-CCD array while the various bits of each word correspond to the past time instances. See Figure 3.7. Bit 0 is always set to zero as will be explained later. Bit 1 of each word represents the threshold violation information of the current packet, bit 2 corresponds to one packet into the past, bit 3 corresponds to two packets into the past, etc. This format is used to simplify the target-no target decision process.

The computational routine, called "crunch", is written in LSI-11 assembly language and may be broken down into three main parts. These are (i) the computation of the background spatial average, (ii) the computation of the background time average, and (iii) the threshold comparison and decision making section. The background spatial average is computed by summing D equally spaced samples from the current packet of 256 samples and then dividing by D. In the software implementation of this calculation, there are three auxiliary variables that are defined in "C" routines to speed the routine up. "ntbkspav" is set to equal the number of terms used in computing the background spatial average (256/D) and is used as the initialization for a counter to keep track of how many iterations are needed in this computation. "tbgd" is set equal to 2*D and is used for stepping along the array

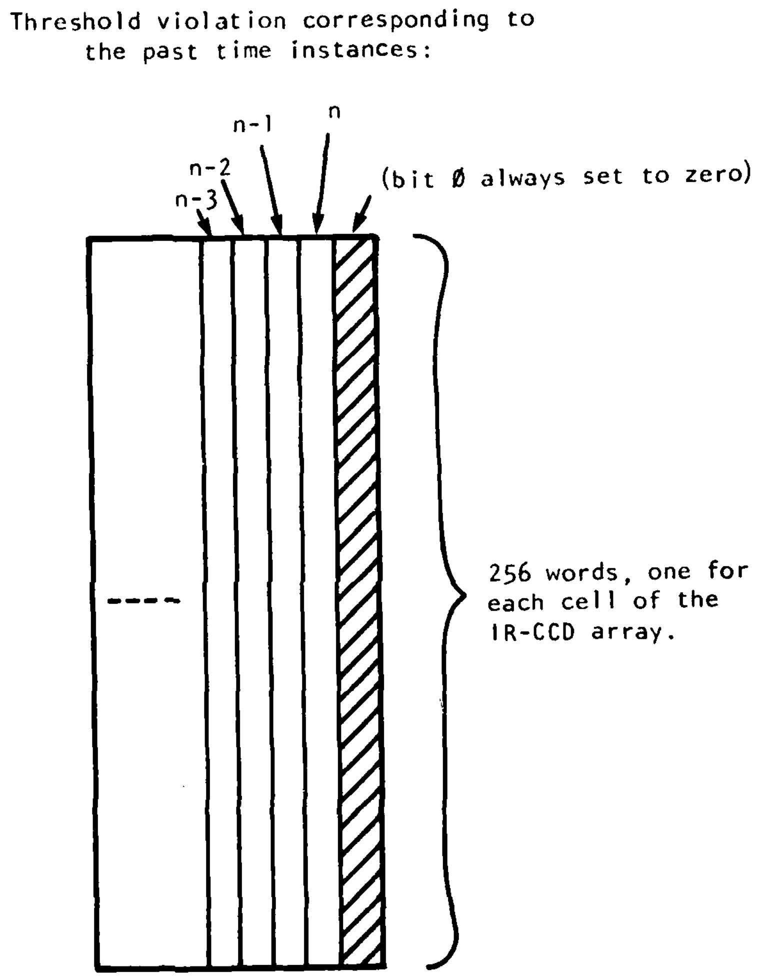

Threshold violation corresponding to the past time instances:
n-3
n-2
n-1
n
(bit Ø always set to zero)
----
256 words, one for each cell of the IR-CCD array.

Figure 3.7 Layout of "a" vector.

of new samples (pointed to by "yptr"). The factor of two is due to the word length of the LSI-11 being two bytes and thus, since each integer value is stored in one word, it is necessary to increment by two to reach successive elements in memory. For example, "yptr" points to the first element in the array of new samples. "yptr" plus "tbgd" points to the Dth element in this array, "yptr" plus 2*"tbgd" points to the 2*Dth element in the array, etc. The section of code that computes the sum of D terms uses this procedure for extracting every Dth element from the packet of new samples. The sum is carried out in double precision to prevent overflow. The third auxiliary variable "dshft" is used in dividing this sum by D. Since all possible D values are powers of two, the division is accomplished by a shifting operation. "dshft" tells the assembly language shifting instruction "ashc" how many places to shift and which direction (negative numbers to the right). At this point in the calculations, the background spatial average is in r3. A flow chart showing the computation of the background spatial average and stare time changing operation is shown in Figure 3.8.

The first thing that is done in checking for a store time change is to determine whether or not a stare change was performed during the previous set of calculations. If a change was indicated, it hasn't had time to affect the current packet and it would make no sense to check for a change again. The status of whether or not a change took place on the previous packet is indicated by the flag "fstat".

Stare time changes are initiated by comparing the background spatial average with two thresholds, one to indicate a decrease in store time is needed (C2) and one to indicate a stare time increase is in order (C1). These thresholds are not constant for different stare times. When operating at tref, the incoming samples are roughly in the range 256 to 511. When the

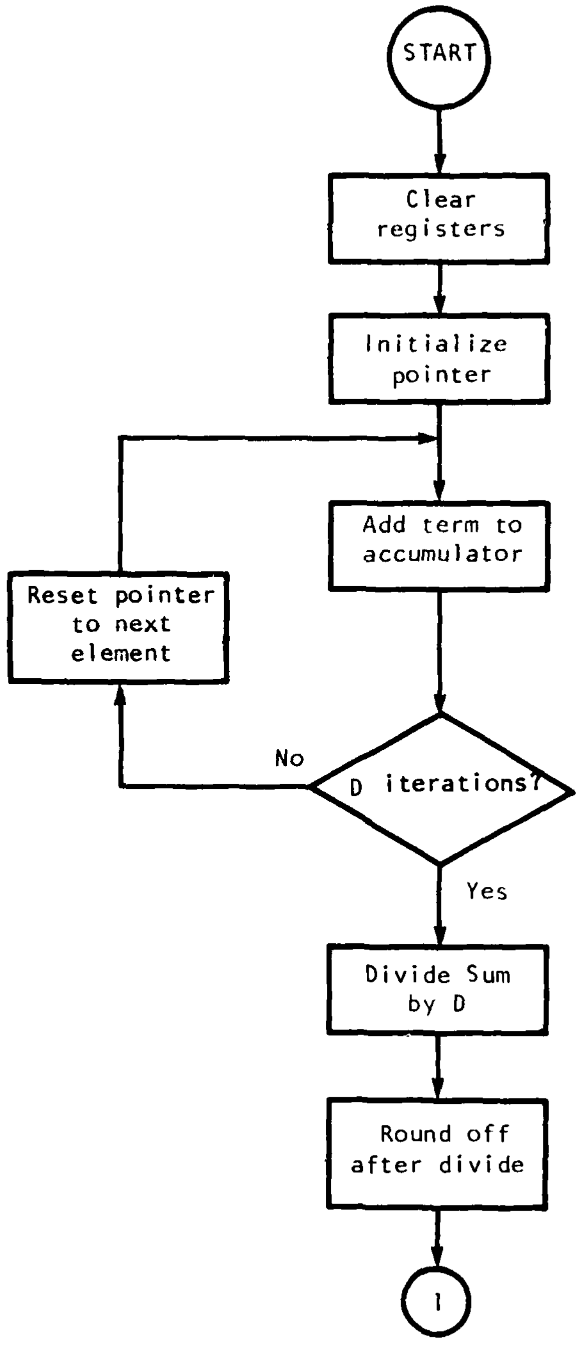
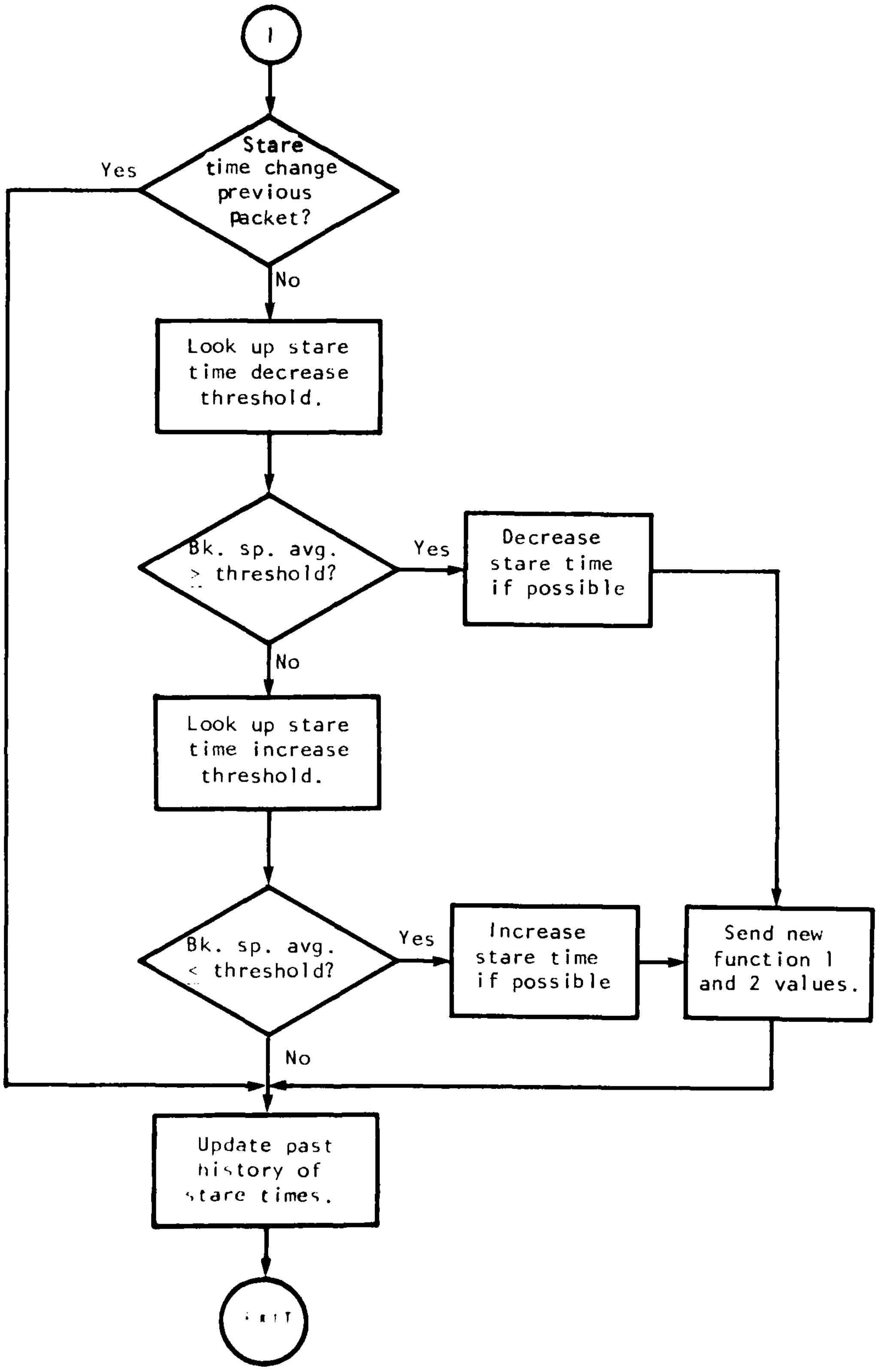


Figure 3.8 Computation of background spatial average and stare time changing operation.

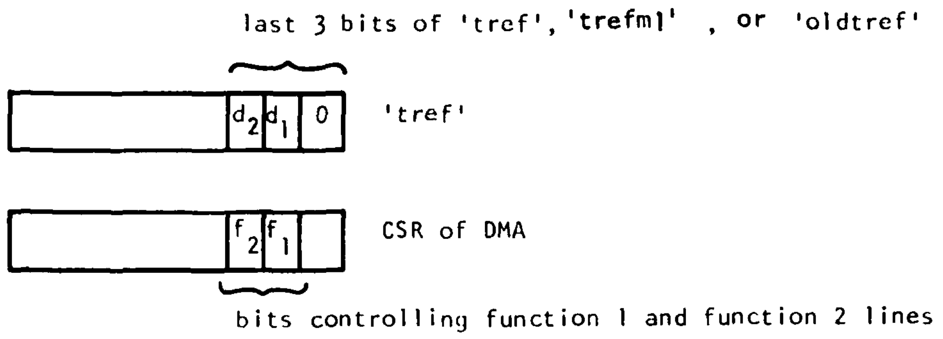

1
Stare time change previous packet?
Yes
No
Look up stare time decrease threshold.
Bk. sp. avg. > threshold?
Yes
Decrease stare time if possible
No
Look up stare time increase threshold.
Bk. sp. avg. < threshold?
Yes
Increase stare time if possible
Send new function 1 and 2 values.
No
Update past history of stare times.
EXIT

[illegible] ard.

stare time decreases to tref/2, the incoming samples are roughly in the range 512 to 1023. Similarly, at tref/4 the samples range from 1024 to 2047, and at tref/8 from 2048 to 4095. Therefore, as tref decreases, the thresholds must be increased by a factor of two at each change. Similarly as tref increases, the thresholds must be decreased by a factor of two at each change.

Since the current samples were integrated by the IR-CCD array with the stare time computed two packets before, it is necessary to store this past history of stare times. "tref" holds the value that will be computed for the current packet, "trefm1" holds the value computed one packet into the past, and "oldtref" holds the value computed two packets into the past. The values that are stored serve several purposes. These purposes and the possible values for "tref", etc. are illustrated in Figure 3.9. This shows how "tref" may be used to change the function 1 and 2 lines and how "oldtref", the value of "tref" from two packets ago, may be used as a pointer to look up elements in the arrays "CC1" and "CC2" which hold the properly scaled values of the user specified parameters C1 and C2.

When it is determined that one of the thresholds has been violated and that a stare time change is in order, the current value of "tref" must be checked to see if it is possible to increase or decrease the stare time any more. If a change cannot be made, "tref" is saturated at the appropriate extreme value. If a change is possible, "tref" is updated by adding or subtracting 2 from "tref". This new value is then sent out on the DMA function 1 and function 2 lines and the past history of stare time changes is updated by moving "trefm1" to "oldtref" and "tref" to "trefm1". This completes the section of code which computes the background spatial average and initiates a stare time change when needed.


last 3 bits of 'tref', 'trefm1' , or 'oldtref'
d_2
d_1
0
'tref'
f_2
f_1
CSR of DMA
bits controlling function 1 and function 2 lines

stare time	d_2	d_1	f_2	f_1	decimal value of d_2 d_1 0
tref	1	1	1	1	6
tref/2	1	0	1	0	4
tref/4	0	1	0	1	2
tref/8	0	0	0	0	0

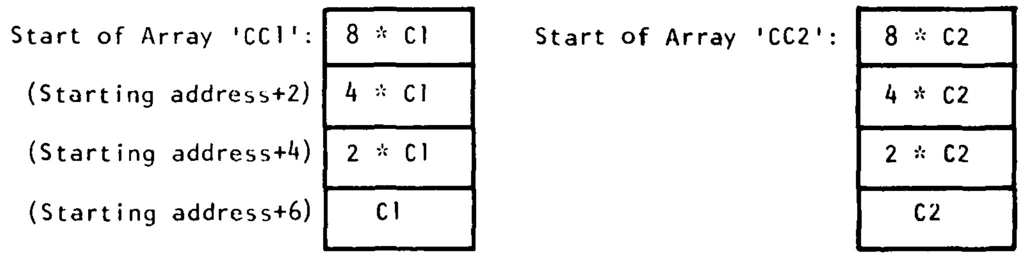


Figure 3.9 Uses of stare time information.

The second major part of the calculations involves computing the background time average and looking up in a table the decision threshold that corresponds to this time average. There are a total of 256 background time averages that must be computed, one for each cell of the IR-CCD array and N terms used in computing each average. Over the course of P packets, all 256 averages are updated by adding in one new term and subtracting out the oldest term. In order to save time, only one P^{th} of the cells are updated each packet. There are three auxiliary variables defined in the "C" routine "ppsrv" which are used to divide the calculations over P packets. These are "n1", "pnorm", and "plast". These numbers are computed so as roughly to divide the computations evenly over each iteration. "n1" indicates the total number of passes during which any background time averages are updated. For most values of P, "n1" will be equal to P. However, for some P values, it is more efficient in terms of reducing the maximum number of cells that are updated during any single pass (worst case condition) to have "n1" less than P. "pnorm" is the number of cells that are updated during the first "n1" minus 1 passes, while "plast" indicates the number of cells updated during the "n1"th pass. There are no cells updated during the remaining P minus "n1" passes. For example, with P equal to 12, "n1" is 12, "pnorm" is 21 and "plast" is 25. This means that during the first "n1" - 1 or 11 passes, 21 cells are updated while during the 12th pass, the remaining 25 cells are updated.

Since the computations are distributed like this, it is necessary to store the location of the next cell to be updated in the array of 256. This is accomplished by storing three pointers. "xptr" points to the next location in the previously described "x" array which holds past sample values, "mptr" points to the next location in the previously described "m" vector which holds the accumulated sum of terms used in computing the background time average, the background time average itself, and the corresponding decision thresholds, and "yincr" is used to point to the next location in the current array of new samples whose start is pointed to by "yptr". It is pointed out again that due to the addressing structure of the LSI-11 processor, it is necessary to increment by two to point to successive words in memory. This factor is automatically accounted for in the various auto-increment, auto-decrement instruction modes [1].

The calculation of the background time average begins by incrementing a counter "pcnt" which is used to keep track of which block of the 256 IR-CCD cells is to be worked on. Also performed at this time is the loading of the various pointers into registers. These pointers indicate where the next cell is to be updated in the "x", "m", and "y" arrays. As mentioned before, "x" contains the past samples used in computing the background time average, "m" contains the sum of N of these samples for each cell, the actual background time average, and the corresponding threshold, while "y" (actually "ybuf1" or "ybuf2") is the array of 256 current samples. The pointer to the "x" vector is checked to determine if the last element in the vector has been reached and if so, is reset to point back to the first element. This is where the previously mentioned circular nature of the "x" vector is accomplished. The value of "wrap" is used to determine when the end of the "x" vector is reached. "wrap" is set equal to 2*N*256 (the two factor is again required in order to skip words instead of bytes).

Now that all is set up for the calculations to begin, it must be determined which segment of calculations is to be performed. If "pcnt" is less than "n1", program control is transfered to a section of code that performs the calculations on the next "pnorm" cells of the array. If "pcnt" is equal to "n1", program control is transfered to a section of code that performs the calculations on the final "plast" cells of the array. When all 256 cells have been updated, "pcnt" is compared to P (stored in "bigp") to determine if it is time to start updating the entire block of 256 cells again. If "pcnt" is equal to P, "pcnt" is set equal to zero and the procedure is started over from the beginning at the next pass through the computations. This procedure for distributing the background time average calculations over P packets is illustrated by the flow chart of Figure 3.10.

The actual computation of the background time average and determination of the corresponding decision threshold for one cell of the IR-CCD array will now be discussed. At the beginning of a typical iteration, r0 is pointing to the proper element of the current data packet, r1 is pointing to the corresponding element in the first block of 256 words of the "m" vector. It is noted that $(1000)_8$ + r2 points to the corresponding element of the second 256 words of "m" (the background time average), $(2000)_8$ + r2 points to the corresponding element of the third 256 words of "m" (the decision thresholds corresponding to the background time averages in the second 256 words), and $(3000)_8$ + r2 points to the last 256 words of "m" (the high order word of the double precision sum of past samples). The first thing that is done in these calculations is to subtract (in double precision) the oldest sample stored in the "x" vector from the sum of past samples stored in the first (and last) 256 word block of "m". Next, the newest sample, pointed to by r0, is added (double precision) into the accumulated sum pointed to by r2. This double precision sum is then moved into registers r4 and r5 and the fixed point division by N is performed using the "div" instruction. After rounding off after the division, the single precision background time average is stored in the proper element of the second 256 word block of "m".

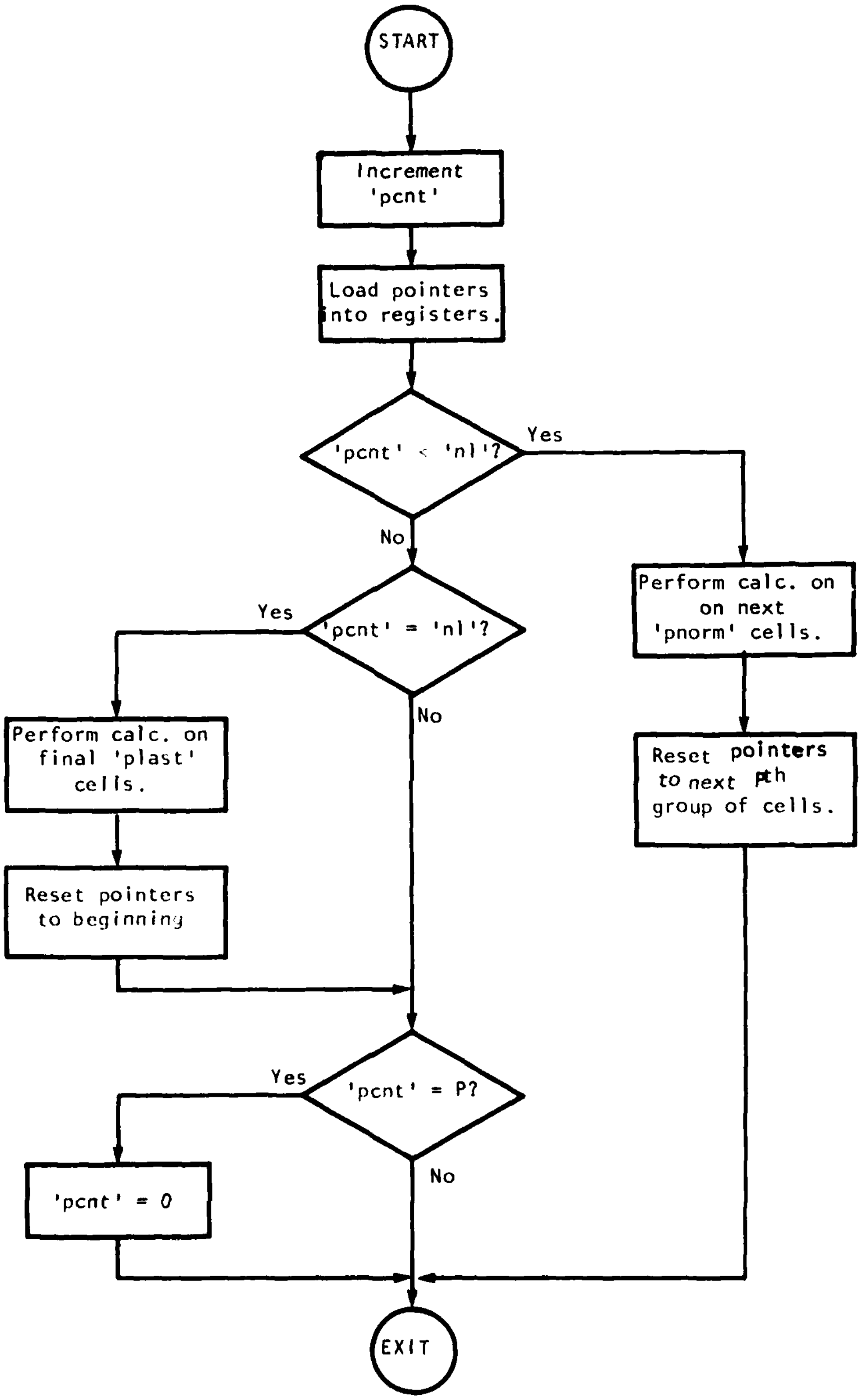

START
Increment 'pcnt'
Load pointers into registers.
'pcnt' < 'nl'?
Yes
No
Perform calc. on on next 'pnorm' cells.
Reset pointers to next pth group of cells.
'pcnt' = 'nl'?
Yes
No
Perform calc. on final 'plast' cells.
Reset pointers to beginning
'pcnt' = P?
Yes
No
'pcnt' = 0
EXIT

Figure 3.10 Distribution of calculations over P packets.

In the "dsrv" and "nnsrv" routines, the assembly language routine "thresh" is called. This routine sets up a table of decision thresholds for the current values of N and d where each element in the table corresponds to a different background time average ranging from 0 to 1023. For background time averages greater than 1023, it is noted that dividing the average by 4, looking up the corresponding threshold, and multiplying this result by 2 (since the thresholds are proportional to the square root of the background time average) effectively extends the range of the table from 0 to 4095. It is also noted that when the stare time is equal to "tref" or "tref"/2, samples will range from 0 to 1023 and when the stare time is equal to "tref"/4 or "tref"/8, incoming samples will be in the range of roughly 1024 to 4095. Thus, when the stare time is equal to "tref" or "tref"/2, the threshold corresponding to the background time average is directly looked up in the threshold table. When the stare time is equal to "tref"/4 or "tref"/8, the alternate look up procedure is employed.

Since word addresses must be used to find elements in the table, an additional factor of two is found in the code implementing the look up operation. For the two longer stare time, 2 times the background time average is added to the starting address of the threshold table to find the corresponding threshold. For the two shorter stare times, the background time average is only divided by two (instead of four) before it is added to the starting address of the threshold table to find the corresponding threshold. This threshold is then multiplied by two to get the actual threshold value. Once the threshold is found for any of the various stare times, it is stored in the proper element of the third block of 256 words of the "m" vector for future use in the target decision making process. The computation of the background time average and corresponding decision threshold for a single cell of the IR-CCD array is shown in Figure 3.11.

The final segment of calculations involves performing the target-no target decision process outlined in [2]. The difference operation and threshold comparison operation that is performed on each of the 256 elements in the IR-CCD array for each new packet of data is done simultaneously with the actual target-no target decision. This portion of the calculations is where the previously described "a" vector comes into play.

The main loop of this segment of calculations iterates over each cell of the array. The first step in each intration is to compute the difference between the background time average stored in the second 256 words of "m" and the array of current data samples pointed to by "yptr". The absolute value of this difference is then taken and this value is compared to the previously computed decision threshold which had been stored in the third 256 word block of "m". If the threshold is not violated, the bits in the word of "a" corresponding to that cell are shifted one place to the left and a zero moved into the second bit. (As discussed previously, this corresponds to threshold violation information for the current packet.) If the threshold is violated, the bits in the word of "a" corresponding to that cell are also shifted one place to the left but in this case a one is placed in the second bit. This is the procedure that updates the past history of threshold violations for each cell in the array. The shifting operation allows the information stored during the previous packet now to be moved an additional packet into the past. It is now clear that the target-no target decision process merely involves summing the number of ones contaned in three adjacent words of "a" (corresponding to three spatially adjacent cells of the IR-CCD array), only looking at the bits in each word of "a" that correspond to the n/3 most recent packets.

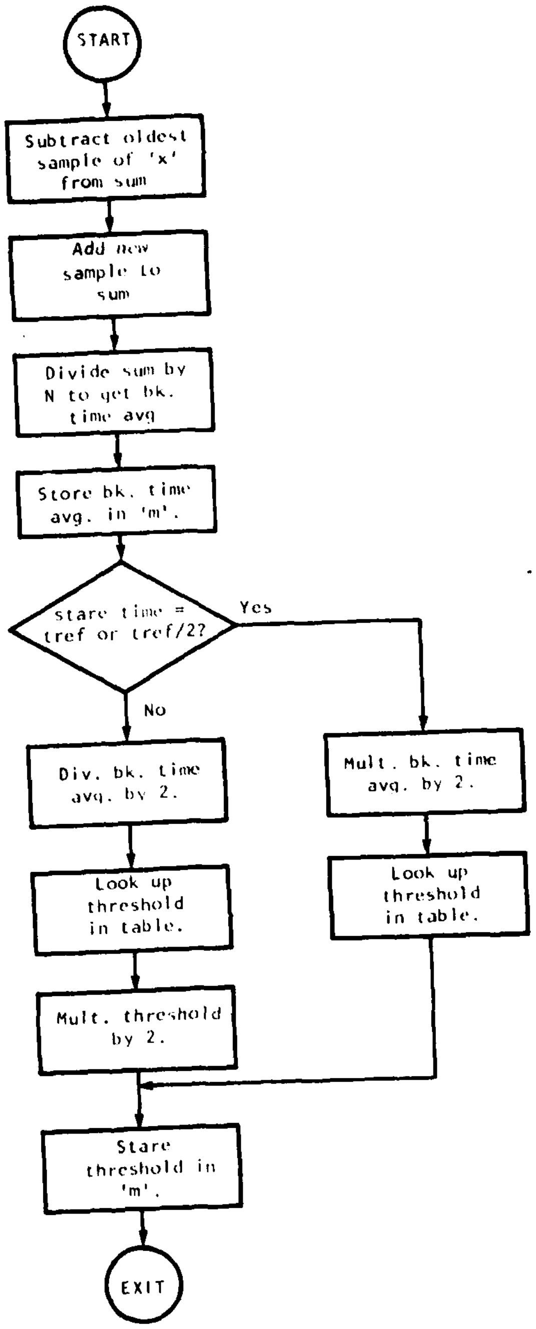

START
Subtract oldest sample of 'x' from sum
Add new sample to sum
Divide sum by N to get bk. time avg
Store bk. time avg. in 'm'.
stare time = tref or tref/2?
Yes
No
Div. bk. time avg. by 2.
Look up threshold in table.
Mult. threshold by 2.
Mult. bk. time avg. by 2.
Look up threshold in table.
Stare threshold in 'm'.
EXIT

Figure 3.11 Computation of background time average and corresponding decision threshold.

This brings up the use of the 1st bit of each word of "a" that is always set to zero. When the bits corresponding to the n/3 most recent packets are masked off, a number results in that word of "a" which can have at most 32 distinct values (when n equals 5). The 0 bit allows this word in "a" to be used as an address to an element in a table whose value is equal to the number of ones (threshold violations) in that cell of the array during the past n/3 most recent packets. This is the procedure employed here.

The actual implementation of this decision process involves the use of the stack (pointed to by sp) and an accumulator r5. For any iteration through this section of code, r5 contains the sum of the number of threshold violations in the previous two cells (this is why no alarm decision is made for the first two cells). The number of threshold violations for the current cell is found by table look-up. This number is stored on the stack (to enable subtraction from the total number of ones for two cells ahead) and also added to r5. Register 5 now contains the total number of threshold violations for three spatially adjacent cells over the n/3 most recent packets. This sum in r5 is then compared to t. If the sum in r5 is greater than or equal to t, a flag (called "tar") is set indicating a target has be detected. If the sum in r5 is less than t, the number of threshold violations two cells before is subtracted from the sum in r5. This set of calculations is then repeated until all 256 cells have been completed. If no target is detected during any of these iterations, the flag "tar" is left set to zero. This completes the computations. Figure 3.12 illustrates this target-no target decision process.

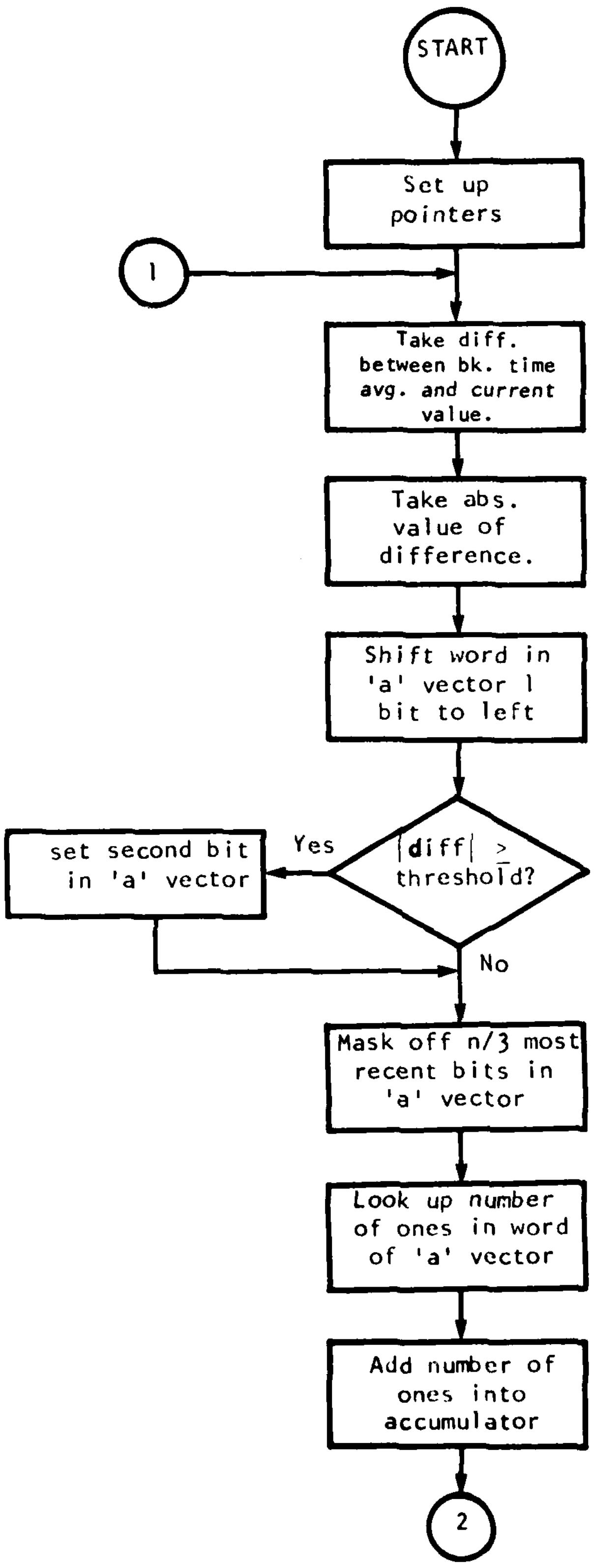

START
Set up pointers
1
Take diff. between bk. time avg. and current value.
Take abs. value of difference.
Shift word in 'a' vector 1 bit to left
|diff| ≥ threshold?
Yes
set second bit in 'a' vector
No
Mask off n/3 most recent bits in 'a' vector
Look up number of ones in word of 'a' vector
Add number of ones into accumulator
2

Figure 3.12 Target/No target decision process.

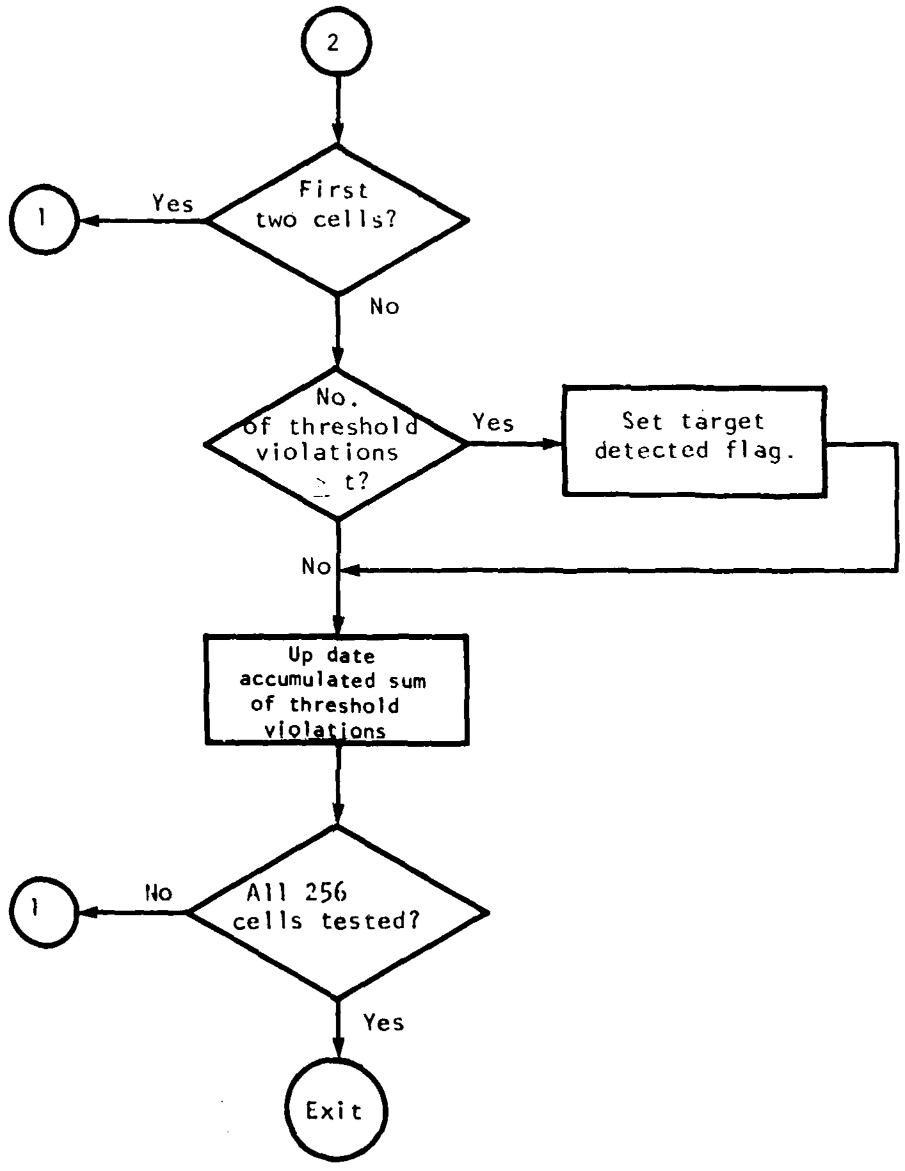

2
First two cells?
Yes
1
No
No. of threshold violations ≥ t?
Yes
Set target detected flag.
No
Up date accumulated sum of threshold violations
All 256 cells tested?
No
1
Yes
Exit

Figure 3.12 continued.

C. *Memory Organization*

There are a total of 16896 words (33792 bytes) of MOS-RAM and 8192 words 16384 bytes) of EPROM contained in the IR-CCD Intrusion Detection System. 16384 of the RAM words are contained on the MSV11-DC board while the remaining 512 words are equally divided on the two MRV11-BA uv PROM/RAM boards. There are 4096 words of PROM on each of the two PROM boards.

The partitioning of memory between RAM and PROM is illustrated in Figure 3.13. The 256 words of RAM from one of the PROM boards starts at address 0. This block of memory provides space for interrupt vectors, trap vectors, etc. At this point there is a 3840 word gap in the addressing of memory locations. This allows PROM to start at a 4k word-multiple. The lower 4k words of PROM start at address $(20000)_8$ while the upper 4k words follow immediately after at address $(40000)_8$. The 16k words of RAM on the MSV11-DC start at address $(60000)_8$ while the remaining 256 words of RAM from the second PROM bound start at address $(160000)_8$.

The locations of several of the more important arrays in the calculations are shown in Figure 3.14. These arrays are discussed in more detail in Sections III.A and III.B but will be briefly summarized here. "ybuf1" and "ybuf2" are input buffers which contain data from the two most recent packets. "x" is the vector that contains past data used in computing the background time average. The "m" vector contains several different quantities. The first and last blocks of 256 words in "m" contain the low and high order words respectively of the sum of all the terms in the background time average. The second 256 word block of "m" contains the actual background time average while the third 256 word block contains the decision thresholds corresponding to the background time averages stored in the second 256 word block. The "a" vector contains the past histories of thres-

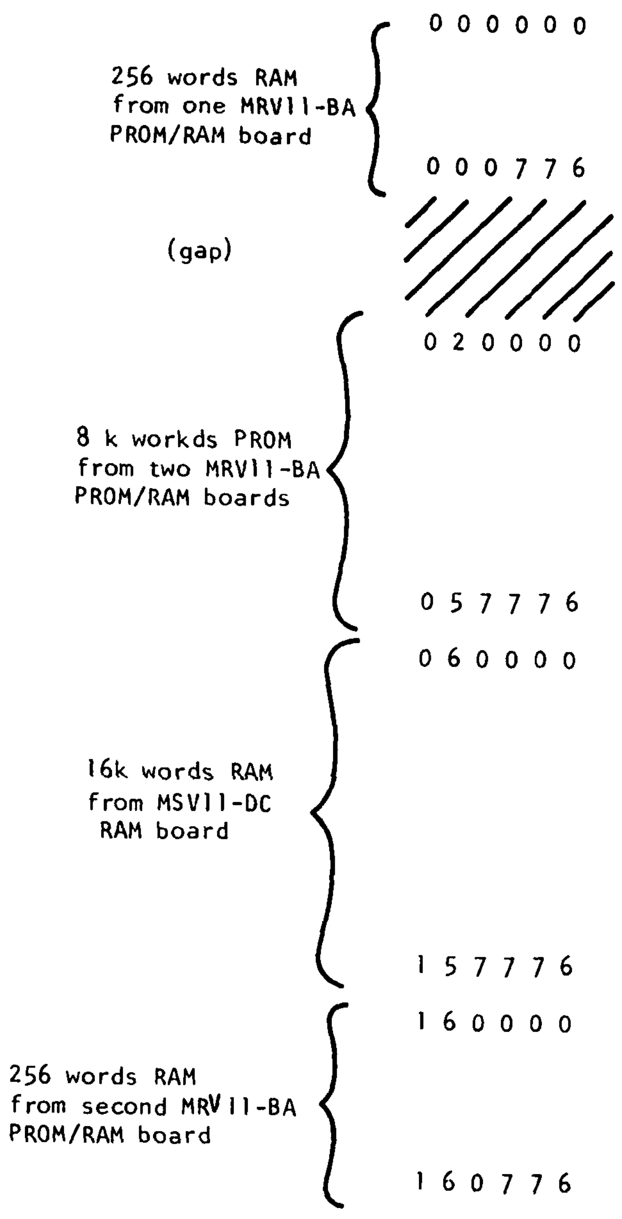

0 0 0 0 0 0
256 words RAM from one MRV11-BA PROM/RAM board
0 0 0 7 7 6
(gap)
0 2 0 0 0 0
8 k workds PROM from two MRV11-BA PROM/RAM boards
0 5 7 7 7 6
0 6 0 0 0 0
16k words RAM from MSV11-DC RAM board
1 5 7 7 7 6
1 6 0 0 0 0
256 words RAM from second MRV11-BA PROM/RAM board
1 6 0 7 7 6

Figure 3.13 Partitioning of memory between RAM and PROM.

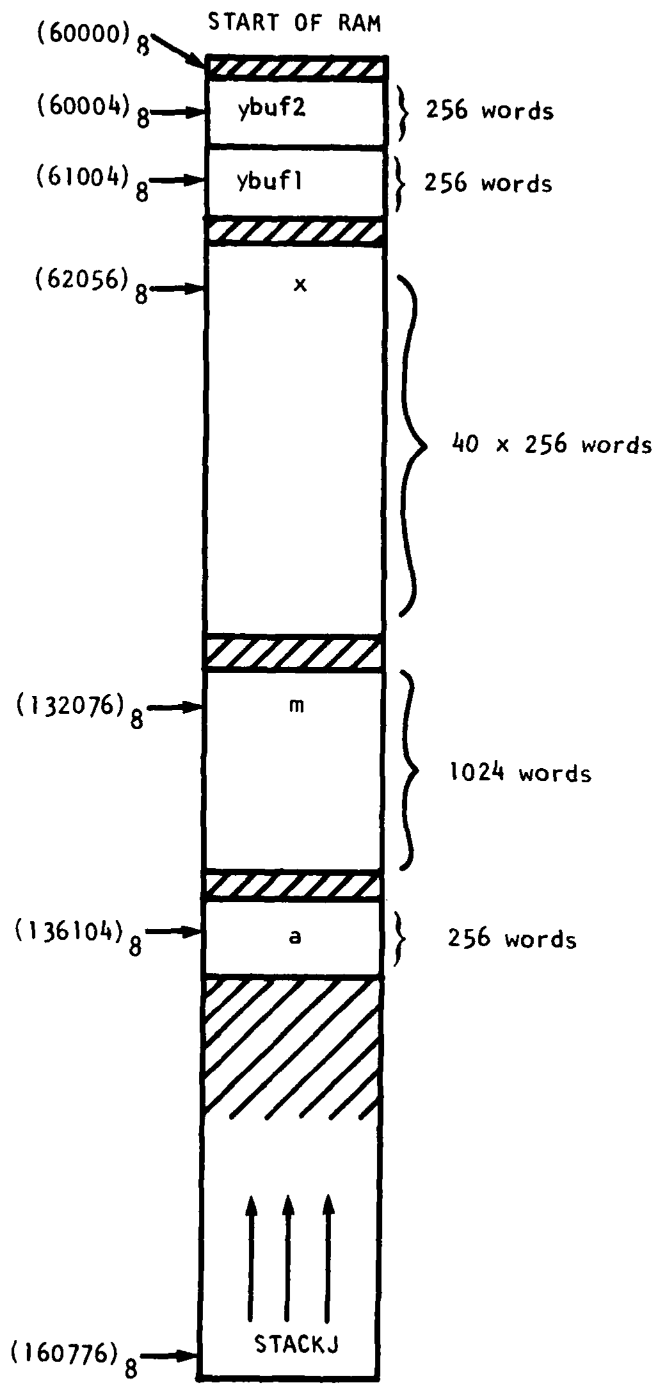


Figure 3.14 Locations of several important arrays in RAM.

hold violation information for each of the 256 cells of IR-CCA array. The PDP-11/03 stack starts at the high end of RAM and sequentially fills up words in descending address order. The stack is used in the calculations and also for keeping track of subroutine calls and passing parameters.

The symbol table contained in Appendix B is useful for probing around inside memory (using ODT) to obtain current values of various variables. Use of this table is best illustrated by a couple of examples. If the current value of the variable "pcnt" were desired, the user would simply find the entry "_pcnt" in the table. (The "C" compiler automatically prepends an underscore to the start of all variable names.) To the left of this name is the number "62052". This is the address in octal where the value of "pcnt" is stored. This location may be accessed via ODT as explained in [1]. Similarly, if it were desired to examine the entries in the array "btab", one would find the address to the left of the "_btab" entry in the symbol table. This number (137122) is the address (in octal) of the first entry in the array. The second entry would be stored in location 137124, the third in 137126, etc. As before these entries may be examined using ODT.

IV. Maintenance

A. Preliminaries to Testing

To shut down power while the system is operating, disconnect the analog CCD signal line, disconnect the 16-pin connector at the IR-CCD device and switch power to "off" at the right rear of the card cage. To power up, simply reverse the order.

To use the card extender(s), shut down power as above, remove board to be inspected, insert card extender with finger "notch" to the left. Insert board into card extender (you should provide support to the rear of the board). Power up as above. <u>When the interface board is to be inspected, removal of the DMA board below the interface is required due to the thickness of the interface board.</u> After DMA and Interface removal, the DMA should be inserted for proper testing to occur. You <u>do not</u> have to disconnect the ribbon connector between interface and DMA in order to remove either board. Board locations in the card cage itself are shown in Figure 4.1.

B. Calibration Procedures and Use of Maintenance Software

The only times the card extender need be used are when a problem is suspected in a board or when calibration of the A/D and sample-and-hold gain, offset, and input voltage range is required. The A/D converter has two input voltage ranges: 0 to +5 volts, and 0 to +10 volts. In order to operate on the 0 to +5 volts range, jumper J1 (pin #25 on the ADC85C-12 unit) must be connected to pin #22 on the ADC85C-12 unit. In order to operate on the 0 to +10 volts range, jumper J1 must be connected to pin A (open). Pin #22 and pin A are both short in length. See Figure 4.2 for location of pins #22, #25 and pin A.

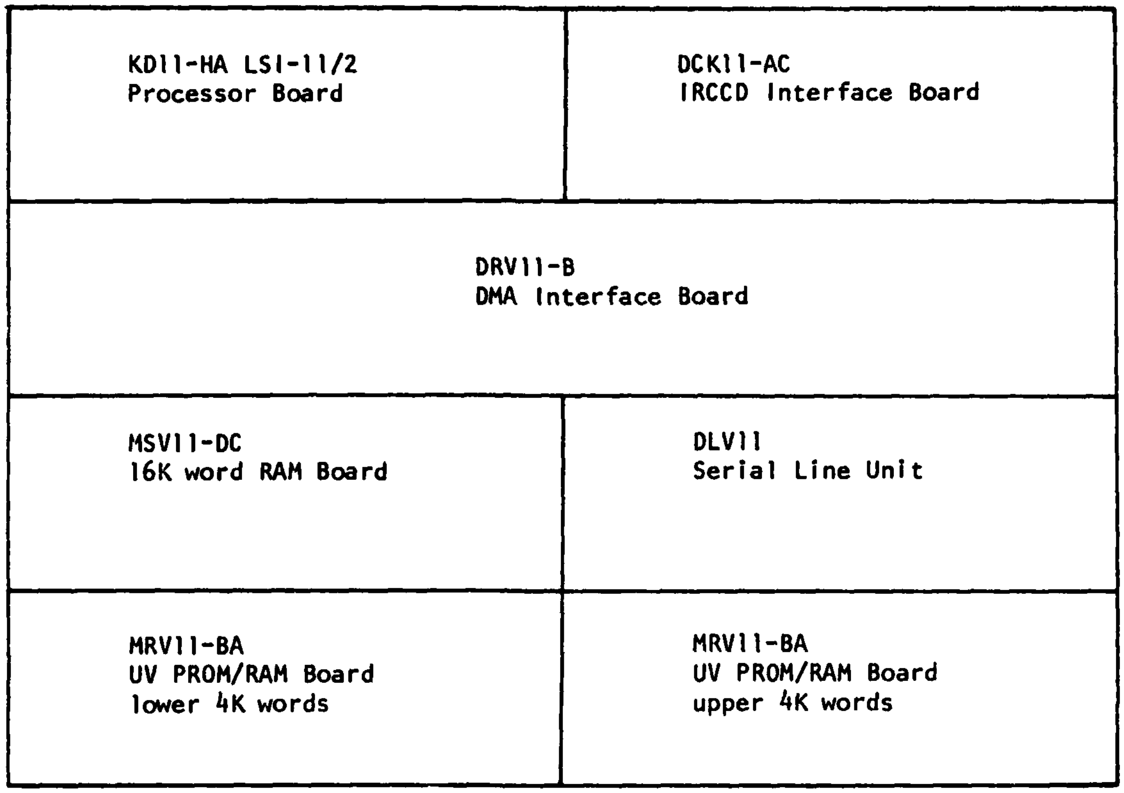


Figure 4.1 Board locations in card cage.

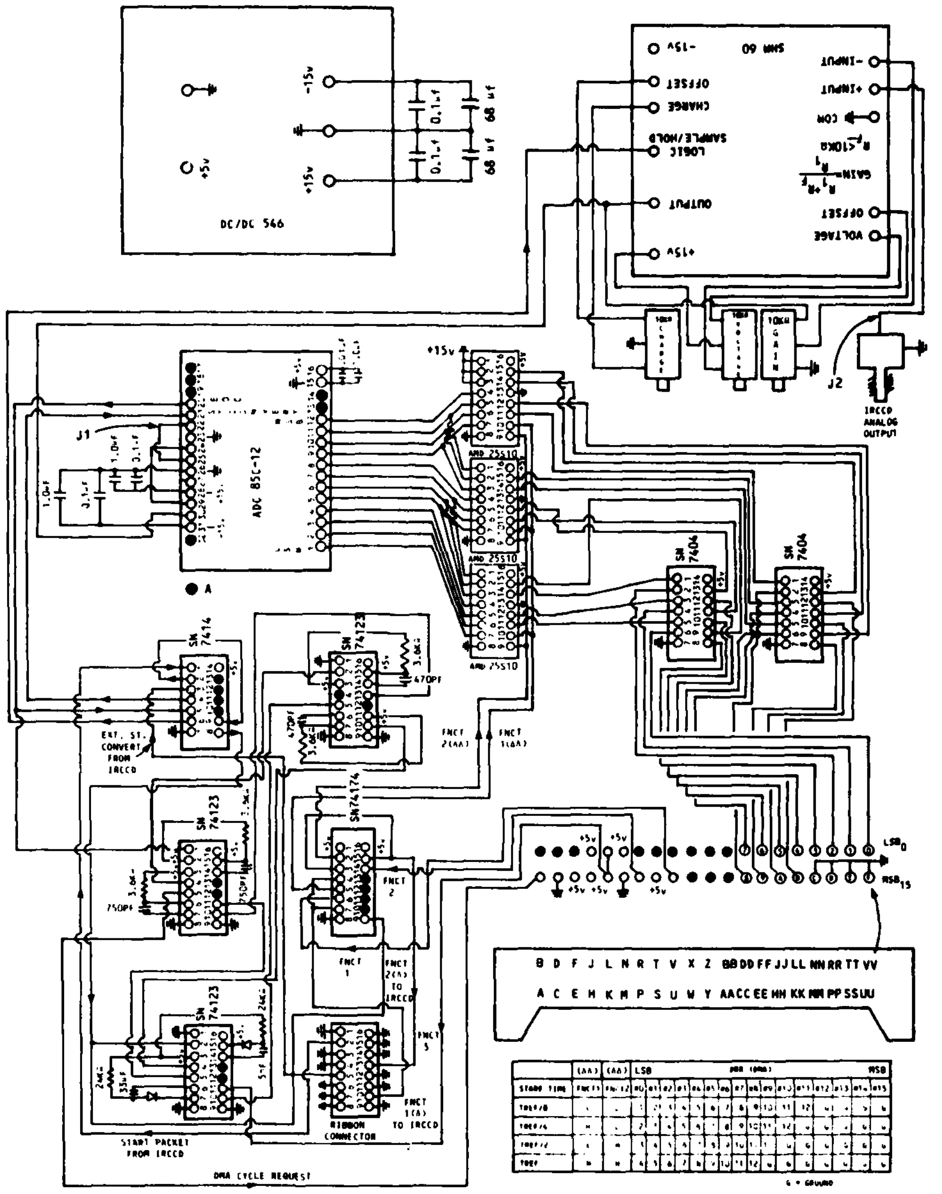


Figure 4.2 Wiring diagram of IR-CCD custom interface.

To adjust the gain of the sample-and-hold device, apply a known dc voltage on the analog line, with the PDP-11/03 powered up. Measure the voltage between "COM" on the SHM60 and Pin #30 on the A/D converter. This voltage is the output voltage of the sample and hold device. Adjust the potentiometer ("pot") marked "GAIN" until the proper output voltage is seen.

There are two types of offset possible: voltage offset, and charge offset. A pot is provided for each. The voltage offset should be adjusted to zero by grounding the analog input, observing the SHM60 output on pin #30 of the A/D converter and adjusting the pot marked "VOLTAGE" until the output voltage indicates zero. The analog signal can be grounded by removing jumper J2 and connecting it to "COM" on the SHM60.

The charge offset is the error between the sample value and the hold value out of the SHM60. This charge offset should be adjusted using the pot marked "CHARGE" so that a zero volt input just barely causes only the least significant bit (LSB) to toggle on and off. To perform this operation ground the analog input as above, examine the least significant bit which is pin #1 of the A/D converter, adjust "CHARGE" until only the LSB toggles. (The charge offset might be considerable in order for this to happen. If you are unable to do this, you can raise the voltage offset away from zero.) Another approach is to examine the entire 12-bit A/D values using the software test that looks at A/D converter output via the terminal. (This is option #2 on the "Additional Features" terminal display frame.) Again ground the analog input and adjust "CHARGE" until only the LSB toggles.

This software test routine can also be used to adjust the overall gain of the SHM60 and the A/D converter so that a known maximum analog input voltage turns on all the A/D converter bits. To do this, simply apply the known maximum analog voltage, enter the software test routine, adjust "GAIN" until all but the LSB are on, and the LSB toggles. Bit #12 (1,2,...,12) will always be low as it is the sign bit.

The use of the various software maintenance routines will now be covered. Each test routine will be discussed individually and possible reasons for test failure will be given. Several of the routines utilize more than one section of the IR-CCD Intrusion Detection System hardware and hence care should be used in diagnosing problems.

(1) A to D Test - This routine displays the output of the A to D converter on the 4024 screen in both binary and decimal. The test works by taking a sample from the A/D converter, setting the function lines to display all 12 bits out of the programmable digital shift (PDS) logic, and writing this sample into a location in the 11/03's memory using a single word DMA transfer. This word is then converted to ascii and displayed on the terminal screen. Hence, failure of the test could imply problems in either the A to D converter, the PDS logic, the DMA board, the RAM board, the serial interface, the terminal, or any of the connections between these devices.

The A to D test is entered by typing a "2" followed by a carriage return when the terminal is displaying the additional features frame. At this point a brief description of the test will appear on the 4024 screen. If the user then types another carriage return, the actual A to D test routine will begin execution. When this happens, there will be two numbers displayed on the screen, one on the upper left portion and one on the upper right portion. The number on the left is the binary value being read from the A to D converter while the number on the right is the decimal value. These numbers are continually updated at a rate of about five to ten times a second depending on the clock frequency. For this test to work, it is necessary for both the "start convert" and "start packet" signals to be connected to the custom interface board. The "start convert" signal initiates the data conversions and also pulses the DMA cycle request line to load the samples into memory. The "start packet" line is used to clock the flip-flops which set the function line values to the PDS logic. The test is terminated by typing any standard ascii key.

Results of the test are dependent on what analog voltage level is being input to the sample-and-hold and how the gain, offset, etc. of the A to D and sample-and-hold are set. When all adjustments are set properly as discussed previously, and the analog input is grounded, the values displayed on the screen should be approximately zero (within about one LSB). As the analog input is increased, the numbers on the screen should increase also until full scale is reached (all 12 bits should be ones). The full scale input voltage level will depend on the gain setting of the sample-and-hold and the input range setting. This full scale saturation test is useful for determining if bits are being dropped somewhere along the way. This condition could be caused by say a broken wire in one of the lines between the custom interface board and the DMA board. Other possibilties exist. The linearity of the A to D conversion process may also be examined using the A to D test routine. When the analog input level is doubled, the number displayed on the terminal screen should also double. Obviously, A/D monotonicity can also be checked this way. This routine is also useful for checking the input level from the IR-CCD array.

(2) Programmable Digital Shift Logic Test - This routine is virtually identical to the A to D test routine except that the function lines are repeatedly cycled through the various possible combinations which in turn cause the PDS logic to perform shifting operations. As before, the output from the PDS logic is displayed on the screen in both binary and decimal.

In addition, the current values of the function 1 and function 2 lines are displayed on the screen. Like the A to D test, there are several different devices involved in the test execution and hence several different places to look for problems if test results are not as expected.

The PDS test is entered by typing a "3" followed by a carriage return when the terminal is displaying the additional feature frame. This causes a brief test description (including a table that indicates how many bits from the A to D converter should be present at the output of the PDS logic) to be displayed on the screen. Typing another carriage return uses the PDS test routine to begin execution. As with the A to D test, there will be two numbers displayed on the screen. The binary representation will be at the upper left while the decimal representation will be at the upper right. In addition, the current values of the two function lines will appear in the center of the screen. The data values displayed on the screen are continually updated at a rate of about five to ten times a second while the function line values change every one to two seconds. Like the A to D test both the "start convert" and "start packet" signals must be applied to the custom interface board. The test is terminated by typing any standard ascii key.

When analog voltages are applied to the analog input line of the custom interface board, the corresponding digital values should appear on the terminal screen. These values are dependent on the A to D/sample-and-hold calibration and the current values of the two function lines. An example of the display values for the various function line settings is shown in Table 4.1 for the case where the A to D is driven full scale. It is seen that changing the function lines changes the digital values by factors of two.

Function Lines		Output of PDS Logic	
f1	f2	Binary	Decimal
0	0	111111111111	4095
1	0	011111111111	2047
0	1	001111111111	1023
1	1	000111111111	511

Table 4.1 Example of output from PDS logic test.

Sometimes it is useful to use this test routine in conjunction with some of the others to help isolate a problem area. For instance, if the PDS test results do not appear correct, the problem may be with the DMA since that device is used to load data samples into memory during the PDS test. Hence, execution of the DMA test routine might aid in isolating the problem. Another case might be that the A to D test appears to work properly but the PDS test fails (the numbers do not shift). This could indicate a break in one or both of the function lines connecting the DMA to the custom interface, a bad flip-flop on the custom interface board (not clocking the function line values to the PDS logic) or possibly the PDS logic chips themselves being bad.

(3) DMA Test - This routine tests seven different facets of DMA operation. The user is referred to [3] for more specific details about the DMA board. A complete discussion of the items tested and the implementation of the test routine is given in Section III.B(4a) in the discussion of the "dma" assembly language subroutine. The description here will discuss the <u>use</u> of this test routine rather than the routine itself.

The DMA test may not be entered directly from the additional features frame due to the need for inserting the DMA maintenance cable. When the additional features frame does appear on the screen the user may type a "4" followed by a carriage return to display a brief summary of the DMA test. When the user hits another carriage return, a PDP-11/03 "halt" instruction is executed and the processor returns to the ODT mode. At this point the following procedure should be executed to perform the DMA test:

(i) Disconnect signals from custom interface board.

(ii) Power down 11/03 processor.

(iii) Partially remove DMA board so ribbon cable connectors are accessible.

(iv) Disconnect both ribbon cables from DMA board.

(v) Insert DMA maintenance cable between two connectors on DMA board.

(vi) Reinsert DMA board into slot.

(vii) Power up 11/03 processor.

(viii) Issue "20000G" sequence as outlined in startup procedures.

One of the first things that the initialization routine does is test if the DMA maintenance cable is in place by sending out values on the three function lines and determining if these values are read back correctly by the three status lines (as will be the case if the maintenance cable is in place). If issuing the "20000G" sequence when the DMA maintenance cable is in place does not start execution of the DMA test routine, check connections of the maintenance cable and try again. If the test is still not entered, it is likely that there is a problem with the DMA.

When the DMA test routine is executed, seven lines will appear on the terminal screen which will indicate the success or failure of each of the seven DMA functions tested. These display messages are shown in Table 4.2. The execution of the test routine is repeated any time a standard ascii key is typed on the keyboard. The test is terminated by hitting the 'break' key rapidly twice in succession. This returns the processor back to the ODT mode. To restart the program for the normal execution again, the procedure outlined for entering the DMA test should be reversed.

1	(Error/No error) detected in R/W of WCR.
2	(Error/No error) detected in R/W of BAR.
3	(Error/No error) detected in R/W of DBR.
4	(Improper/Proper) response received from INIT signal.
5	(Error/No error) detected in function-status lines.
6	End of transfer interrupt (not detected/detected).
7	(Error/No error) detected in data transfer test.

Table 4.2 Possible messages displayed by DMA test.

(4) RAM Integrity Test - This routine tests the read/write capabilities of the roughly 16k words of RAM. This is accomplished by writing unique bit patterns into each memory location and then attempting to read these patterns back. If the patterns read back do not agree with what should have been written in, an error is signaled.

The RAM test is entered by typing a "5" followed by a carriage return when the terminal is displaying the additional features frame. This causes a brief description of the test to be displayed on the terminal screen. When another carriage return is typed, the RAM test routine begins execution.

If no errors are detected during the entire test, a message to that effect is displayed on the screen. Typing any standard ascii character will then terminate the test and cause the computations to be restarted at default parameter values. When an error _is_ detected, the location of the error (in octal) is displayed on the screen and the routine waits for the user to type a command. If the user types a carriage return at this point, the test proceeds until either another error is detected (at which point the routine waits again), or the end of RAM is reached. When the end is reached, the total number of errors detected is displayed on the screen in decimal. The test will then terminate when the user types any standard ascii character and the calculations will be restarted with default parameter values. On the other hand, if the user types an ascii character other than a carriage return after an error has been detected, the routine will stop testing RAM locations momentarily display the accummulated error count up to that point on the screen, and finally restart the target detection calculations from the beginning with default parameter values.

(5) PROM Checksum Test - This routine computes two 16 bit checksums, one for the lower 4k words of PROM and one for the upper 4k words. These checksums are then compared to their expected values (stored in two locations of PROM) and results displayed on the terminal screen. This test is intended only to indicate that there is a problem in one or more of the EPROMS. A more detailed procedure for isolating PROM errors and what to do about them is given in Section IV.C.

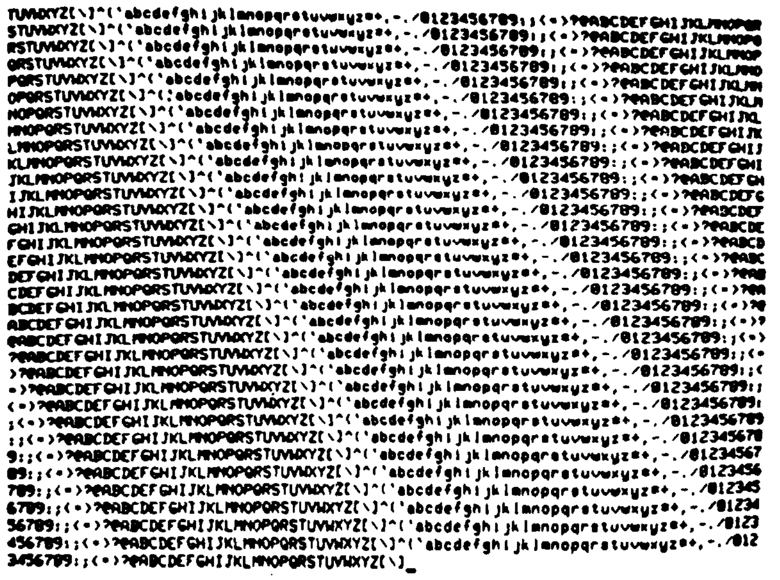
The PROM test is entered by typing a "6" followed by a carriage return when the terminal is displaying the additional features frame. This causes a brief description of the test to be displayed on the terminal screen. When another carriage return is typed, the PROM test begins execution.

When the routine has finished computing the two checksums, four numbers are displayed on the screen. On the left side near the top are the actual and expected checksums for the lower 4k words of PROM while at the right side near the top are the actual and expected checksums for the upper 4k words of PROM. The expected checksum for the lower 4k words is (at this writing) 061224 while the checksum for the upper 4k word block is 174240. In addition to these four numbers, one or two sentences are displayed summarizing the results of the test (whether or not the actual checksums agree with the expected checksums and if applicable which 4k block(s) or PROM are suspect. If errors are detected during this test, it is suggested that the user perform the additional PROM checksum computations outlined in Section IV.C to help isolate which EPROM chip(s) is(are) suspect. That section also contains information about reburning PROMS if necessary.

(6) Terminal/Serial Interface Test - This test is intended to test both the Tektronix 4024 terminal and the DLV11 serial line unit. The routine is divided into three main sections. The first section tests the terminal itself by issuing the standard 4024 "test" command. The serial interface and monitor section of the terminal are then tested by repeatedly sending a line of 81 ascii characters to the monitor section of the terminal. The 81 character length causes the 4024 automatically to issue a carriage return (screen "wrap around") once for each set of 81 characters sent. This in turn causes the line of characters to begin one column to the right each time and hence makes detecting any discrepencies easier. This also allows each of the ascii characters used in the 81 character line to assume every possible position on the screen. The final section of this routine tests the serial interface and workspace section of the 4024 screen in a manner identical to that used for the monitor space.

The terminal/serial interface test is entered by typing a "7" followed by a carriage return when the terminal is displaying the additional features frame. This causes a brief description of the test to be displayed on the terminal screen. When another carriage return is typed, the terminal/serial interface test routine begins execution.

Upon completion of the standard 4024 test routine, a bell is sounded and the screen is left displaying the results of the test. The user is referred to [4] for information on interpreting these results. A prompt is also displayed on the screen instructing the user to type a carriage return to proceed with the terminal/serial interface test. When this is done, the terminal should begin displaying the 81 character ascii lines in the monitor section of the 4024 (the monitor has been defined to be the entire screen). When the visible portion of the monitor is full, the screen will begin scrolling and continue indefinitely in this fashion until the user types a carriage return. This causes the screen to be redefined as entirely workspace (except for the last line which must always be monitor) and the 81 character lines to be again displayed. As with the monitor test, when the visible portion of the screen is full, the display will begin scrolling. This will continue until either a carriage return is typed to terminate the test (and restart the calculations with default parameter values) or the internal memory of the 4024 is full. When the internal memory is full, the terminal stops displaying characters and the screen freezes. When this happens the user should type a carriage return to terminate the test and restart the calculations using default parameter values Figure 4.3 shows a typical view of the terminal screen during either the monitor or workspace portions of the terminal/serial interface test.



```
TUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQR
STUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQ
RSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN
OPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHIJKLM
NOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHIJKL
MNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHIJK
LMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHIJ
KLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGHI
JKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFGH
IJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEFG
HIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDEF
GHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCDE
FGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABCD
EFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@ABC
DEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@AB
CDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@A
BCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=>
?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<=
>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;<
=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:;
<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789:
;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456789
:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./012345678
9:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./01234567
89:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123456
789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./012345
6789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./01234
56789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./0123
456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^(`abcdefghijklmnopqrstuvwxyz*+,-./012
3456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]_
```

Figure 4.3 Typical view of terminal screen during either monitor or work-space portions of terminal/serial interface test.

C. PROMS

There are a total of sixteen 2708 UV erasable PROM chips that comprise the 8k words of PROM used in the IR-CCD Intrusion Detection System. Each chip contains 1k bytes of memory. The MRV11-BA PROM/RAM boards divide the 16 bit PDP 11/03 words into high and low order bytes. In other words, one 2708 chip will contain 1k consecutive low order byes while another 2708 will contain the corresponding 1k consecutive high order bytes. Figure 4.4 shows which PROMs correspond to which section of memory on the two PROM boards. The individual PROMs are also labeled with this information.

When it is suspected that there may be problems with the PROM section of storage, it is possible to compute checksums for each 2708 chip using a routine entered in RAM. The values returned from this routine may then be compared to the original checksums to help isolate the problem chip(s). To enter this routine in RAM, it is necessary to use ODT. The user is referred to [1] for information on how to do this. Table 4.3 lists the code that should be entered starting at address 60000 octal. The assembly language listing of the routine is given in Appendix B.

Execution of this routine is begun by typing the sequence "60000G" when in ODT. When completed, the number 60064 is displayed on the screen and the ODT prompt returns. The results of the test are stored in sixteen consecutive locations starting at address 65000 octal. These results can be examined using ODT. Table 4.4 tells which PROM corresponds to which checksum and also what values these checksums should be if all is working properly.

If it is determined that new PROMs need to be burned, simply order the new 2708's from a Motorola distributor and supply them with the appropriate paper tapes. These tapes have already been set up in the Motorola "S1" format. They are identified according to their starting address and whether

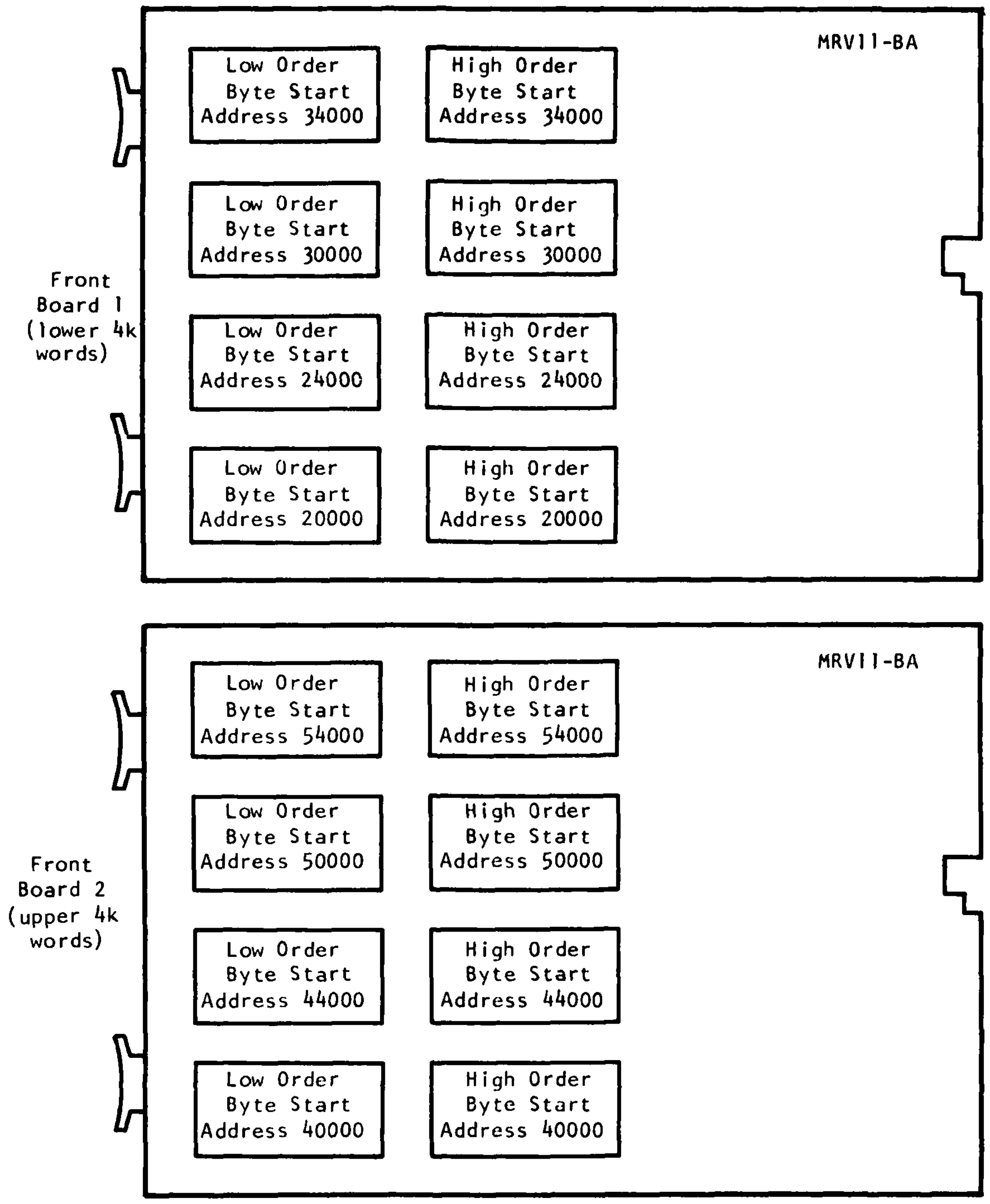

MRV11-BA
Low Order Byte Start Address 34000
High Order Byte Start Address 34000
Low Order Byte Start Address 30000
High Order Byte Start Address 30000
Low Order Byte Start Address 24000
High Order Byte Start Address 24000
Low Order Byte Start Address 20000
High Order Byte Start Address 20000
Front Board 1 (lower 4k words)
MRV11-BA
Low Order Byte Start Address 54000
High Order Byte Start Address 54000
Low Order Byte Start Address 50000
High Order Byte Start Address 50000
Low Order Byte Start Address 44000
High Order Byte Start Address 44000
Low Order Byte Start Address 40000
High Order Byte Start Address 40000
Front Board 2 (upper 4k words)

Figure 4.4 Location of PROMs on boards (top view).

Address	Contents	Address	Contents
60000	012700	60032	002000
60002	000020	60034	111304
60004	012701	60036	060412
60006	065000	60040	005203
60010	005021	60042	111304
60012	077002	60044	060462
60014	012703	60046	000002
60016	020000	60050	005203
60020	012702	60052	077010
60022	065000	60054	062702
60024	012701	60056	000004
60026	000010	60060	077115
60030	012700	60062	000000

Table 4.3 Machine language code for performing checksums on individual PROMs.

Address where Checksum is stored	Starting Address of Corresponding PROM	High or Low Order Byte	Correct Value of Checksum
65000	20000	Low	165502
65002	20000	High	014634
65004	24000	Low	004127
65006	24000	High	165676
65010	30000	Low	161337
65012	30000	High	170301
65014	34000	Low	137434
65016	34000	High	027706
65020	40000	Low	000542
65022	40000	High	045242
65024	44000	Low	031037
65026	44000	High	026626
65030	50000	Low	021106
25032	50000	High	022066
65034	54000	Low	003536
65036	54000	High	004530

Table 4.4 Correct results for checksum test of Table 4.3.

they correspond to the high or low order bytes of the 1k blocks of memory for which they contain information. There are sixteen of these paper tapes, one for each of the sixteen PROMs on the two DRV11-BA PROM boards.

V. REFERENCES

1. Digital Equipment Corporation, "LSI11 PDP11/03 Processor Handbook", Maynard, Massachusetts, 1975.

2. G. R. Cooper and C. D. McGillem, "IRCCD Intrusion Detection", Final Report for Rome Air Development Center, Hanscom AFB, Ma. under Contract No. F30602-75-C-0082, 1977, RADC-TR-77-435, AD# A063 327.

3. Digital Equipment Corporation, "DRV11-B General Purpose DMA Interface User's Manual", No. EK-DRV1B-OP-001, Marlborough, Massachusetts, 1st Edition, August 1976.

4. Tektronix, Inc., "4024/4025 Computer Display Terminal Programmer's Reference Manual", No. 070-2402-00, Beaverton, Oregon, Dec. 1978.

Appendix A.

The following is a list of part numbers with a short word description, manufacturer, and vendor.

Part #	Description	Manufacturer	Vendor
BA11-ME	Card Cage	A	1
H780/E	Power Supply	A	1
7270KD11-HA	Processor Board	A	1
DCK11-AC	Interface Board	A	1
DRV11-B	Direct Memory Access (DMA)	A	1
MSV11-DC	16K Word by 16 bit RAM Board	A	1
DLV11	Serial Line Unit	A	1
MRV11-BA	UV 4K Word by 16 bit PROM/RAM Board	A	1
KEV-11	EIS/FIS Extended Arithmetic Instruction Chip	A	1
2708	Motorola, Intel UV Erasable PROMS	-	-
Am25S10	4-bit shifter TTL chips	C	3
SHM60	high-speed sample-and-hold	B	2
ADC85C-12	high-speed analog-to-digital converter	B	2
DC/DC 546	+5v to ±15v DC-to-DC converter	B	2
	Assorted TTL	-	-

Manufacturers:

A	Digital Equipment Corporation Corporate Headquarters Maynard, Massachusetts 01754 (617) 897-5111
B	Burr-Brown Research Corporation Inc. International Airport Industrial Park P. O. Box 11400 Tucson, Arizona 85734 (602) 746-111
C	Advanced Micro Devices 901 Thompson Place Sunnyvale, California 94086 (408) 732-2400

Vendors:

1	Hamilton-Avnet 954 Senate Road Dayton, Ohio 45459 (800) 543-4783
2	Burr-Brown Research Corporation Inc. 33 North Addison Road Addison, Illinois 60101 (312) 832-6520
3	C. S. Electronics Sales 1157 B South Jackson Street Frankfort, Indiana 46041 (317) (659)-1874

Appendix B.

The software listings and the symbol table are contained in this appendix and are presented in the following order:

(1) Variable Definitions

(2) "C" Routines

(3) Assembly Language Routines

(4) PROM Checksum Routine

(5) Symbol Table

```
/*
*****************************************************************
**                                                             **
** DEFINITIONS - all uninitialized global variables and all   **
** initialized data are declared in this section of code.     **
** These variables are used by both 'C' routines and          **
** assembly language routines. Also included at the start     **
** are 'define' statements which allow certain constants to   **
** be used by inserting the symbolic name given in the        **
** define statement                                           **
**                                                             **
*****************************************************************
*/



/* DEFINE statements */

#define        TTIBUF  0177562 /* address of serial interface input buffer */
#define        TTYSTAT 0177560 /* address of serial interface input status word */
#define        TTYOSTAT 0177564        /* address of serial interface output status word */
#define        TTOBUF 0177566  /* address of serial interface output buffer */
#define        INTVEC 0124     /* address of DMA interrupt vector */
#define        WCR     0172410 /* address of DMA word count register */
#define        BAR     0172412 /* address of DMA bus address register */
#define        CSR     0172414 /* address of DMA control status word */
#define        STL4K   020000  /* address of start of lower 4k words of PROM */
#define        ENDLFK  037776  /* address of end of lower 4k words of PROM */
#define        ENDUFK  057640  /* address of end of upper 4k words of PROM */
#define        STRTRAM 060000  /* address of start of RAM */
#define        ENDRAM  0160776 /* address of end of RAM */



/* Uninitialized data space to be put in RAM */

char           junkin[82];      /* buffer to hold line of chars in tty test */
int            bufnum; /* flag to indicate which input buffer */
int            flag;   /* flag used in DMA test to see if interrupt is received */
int            ibuf1[256];      /* input buffer no. 1 */
int            ibuf2[256];      /* input buffer no. 2 */
int            *iptr;  /* pointer to input buffer */
int            *pptr;  /* pointer to past data vector */
int            p[10240];        /* past data vector */
int            m[1024];         /* sum of past terms, bkgrnd time avg, thresholds */
int            a[256]  /* history of threshold violations */
int            btab[1024];      /* table to hold target thresholds */
int            *mptr;  /* pointer to m vector */
int            *aptr;  /* pointer to a vector */
int            *bptr   /* pointer to btab */
int            pcnt;   /* cntr to indicate which Pth of calcs to be done */
int            ntbkgav;         /* number of terms in background spatial avg */
int            twod;   /* two times D */
int            dshft;  /* no. of bits to shift to divide by D */
int            pstat;  /* status of previous packet stare time changing */
int            cc1[4]; /* array of scaled C1 values */
int            cc2[4]; /* array of scaled C2 values */
int            np;     /* product of N and P */
int            aldisab;        /* number of packets alarm should be disabled */
int          intflg;  /* flag to tell if calcs should be skipped */
int          tref;    /* stare time for current packet */
int          trefm1;  /* stare time one packet into past */
int          oldtref;         /* stare time for two packets into past */
int          wrap;    /* address to wrap around circular queue in x */
int          xincr;   /* counter used to point to location in current input vector */
int          mask;     /* mask to look at past n+2 most recent threshold violation bits */
int          pnorm    /* number of cell the first n1 passes */
int          tpnorm   /* two times pnorm */
int          n1;      /* number of passes computations on pixnd time avg are performed */
int          rdtrsh   /* value used for rounding off after divide */
int          plast;   /* no. of cells updated the n1 pass */
int          tmp1;    /* place for temporarily storing register value */
int          tmp2;    /* place for temporarily storing register value */
int          tmp3;    /* place for temporarily storing register value */
int          sqn;     /* equal to sqrt((N+1)/N) */
int          dmacsr;  /* used to store DMA CSR values */
char         *ttinstat;        /* pointer to serial interface input status register */
char         *ttostat;         /* pointer to serial interface output status register */
char         ttibuf[10];      /* input char buffer */
char         *ttiptrib;       /* pointer to input char buffer for placing chars in buf */
char         *ttiptrob;       /* pointer to input char buffer for removing chars from buf */
int          tticnt;  /* counter of number of chars in input buffer */
int          outstk1[41];     /* output queue for info from program control */
int          outstk2[41]       /* output queue for info from interrupt control */
char         **bfptr1;        /* pointer to elements pointed to by elements of outstk1 */
char         **bfptr2;         /* pointer to elements pointed to by elements of outstk2 */
char         **bfptot1        /* pointer to pointer for removing elements from output queue 1 */
char         **bfptot2;       /* pointer to pointer for removing elements from output queue 2 */
int          btcnt1   /* value indicates which byte of string pointed to by elements of outstk1 */
int          btcnt2   /* value indicates which byte of string pointed to by elements of outstk2 */
int          datareg;         /* temporary location for storing new parameter values */
int          command;         /* holds number corresponding to current command */
int          oldcomnd;        /* holds number representing previous command entry */
int          audalrm;         /* flag indicating whether or not audable alarm is enabled */

/* Locations for storing parameter values */

int          bigd,bign,bigp,smd,smn,smt,c1,c2,sml;

char         datstrng[10];    /* temporary buffer to hold ascii representation of new parameter value */
int          holdcnt          /* counter in char buffer */

/* Locations for storing ascii representations of parameter values */

char         bigdstr[4];
char         bignstr[4]
char         bigpstr[4];
char         smdstr[3];
char         smtstr[4];
char         smnstr[4];
char         c2str[5];
char         c1str[5];

char         atdoct[7];       /* buffer for holding ascii representation of octal number */
char         atdbin[12]       /* buffer for holding ascii representation of binary number */
char         flines[6];       /* buffer for storing ascii representation of function lines */
int          atodtxt;          /* location for storing word from A to D converter */
int          chksm1;  /* checksum for lower 4K of PROM */
int          chksm2;  /* checksum for upper 4K of PROM */
```

```
char           ck1[7]; /* buffer for storing ascii representation of chksm1 */
char           ck2[7]  /* buffer for storing ascii representation of chksm2 */
char           exck1[7];        * buffer for storing ascii rep of expected checksum 1 */
char           exck2[7];        * buffer for storing ascii rep of expected checksum 2 */
char           errlok[7];      /* buffer for storing ascii rep of RAM error location */
char           errcnt[7];      /* buffer for storing ascii rep of RAM error count */
int            oldalfg;        /* flag for storing current status of alarm status box */
int            almflg; /* flag indicating new alarm status */
int            tar;    /* flag set in calculations if target detected */

/* Error flags for DMA test */

int            erWCR, erBAR, erDBR, erRST, erFCT, erINT, erTRN;

int            exrer   /* temporary location for use in DMA test */
char           holdchar[2];    /* temp buffer for echoing characters back to terminal */
char           buf[10];         * command storage buffer */



/* initialized data to be put in PROM   */

char           *string[]      but;     /* initialize pointer to but */

/* prl is a table for which the entries are given by
   prl[i]=sqrt((i+2)/(i+1)) with 10 binary place accuracy */

int            prl[]           {055202 047142 044747, 043616, 043034,
                                042441 042153 041742, 041566, 041440,
                                041331 041235, 041152, 041077, 041031,
                                040770, 040733, 040701, 040652, 040625,
                                040602, 040560, 040540, 040522, 040504,
                                040470, 040455, 040442, 040430, 040417,
                                040406, 040376, 040366, 040357, 040350,
                                040342, 040334, 040326, 040321, 040314},

/* tab is a table whose entries are equal to the number of ones
   in the binary representation of the index for that entry */

int            tab[]           {0, 1, 1, 2, 1 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
                                1, 2, 2, 3, 2 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5},

/* cmdno contains a list of numbers corresponding to the
   various command mnemonics  */

int            cmdn []         {1, 2 3, 4, 5 5, 7, 8, 9, 5, 6, 7, 9, 0},

/* cmds is an array that is initialized so that each row contains
   one of the parameter mnemonics */

char           * mds[]          "D" "N" "P" "d", "n", "t" "C2", "C1", "L", "p", "T", "c", "1", 0},

/* The remaining definitions of initialized data are for setting
   up the various ascii strings that are used to construct the
   different display frames of the IRCCD Intrusion Detection
   System  These will not be commented individually
*/

char           blank[]          ""           \004"},
char            f9[]            "'lea f9 /@/,");
char            f10[]           "'lea f10 /@/,");
char            ctrh[]          "'lea 09 /@/,");
char            ctrv[]          "'lea 24 /@/,");
char            bell[]          07,0);
char            crlf[]          015,012,0
char            bksp[]          010,040,010,0
char            ddloc[]         "'wor h,\0014 16,     \010\010\010\010");
char            nnloc[]         {"'wor h,\0016 16,    \010\010\010\010");
char            ppnloc[]        {"'wor h,\0018 16,    \010\010\010\010");
char            dloc[]          {"'wor h,\00110 15,    \010\010\010\010");
char            nloc[]          "'wor h,\00112 16,   \010\010\010");
char            tloc[]          {"'wor h,\00114 16    \010\010\010");
char            c2loc[]         {"'wor h,\00116 16,    \010\010\010\010");
char            c1loc[]         "'wor h,\00118 16      \010\010\010\010");
char            ddlst[]         {"'ris 17 Possible values for D are 16,32  D= \004");
char            nnlst[]         {"'ris 17 Possible value  for N are 2<=N<=4  N=?\004");
char            pplst[]         "'ris 17 Possible value  for P are 10<=P<=256  P=?\004");
char            dlst[]          "'ris 17 Possible value  for d are n nn, 0<d<10  d=?\004");
char            nlst[]          "'ris 17 Possible  alue  for n are 6,9,12,15,n<=t  n=?\004");
char            tlst[]          {"'ris 17 Possible value  for t are 3-15,t<n  t=?\004");
char            c2lst[]         {"'ris 17 Possible value  for C2 are 512<C2,2*C1  C2=?\004");
char            c1lst[]         {"'ris 17 Possible value  for C1 are <1.5*  C1=?\004");
char            dd11[]          {"'wor h,'era m,'era w,\0011 26,IRCCD INTRUSION DETECTOR ALARM");
char            dd12[]          {"\012\015*\117");
char            dd13[]          {"\012\015| SYMBOL  |  VALUE  | MEANING\0013 79,|\012\015");
char            dd14[]          "|     D    |\0014 21,| Number of cells monitored for stare time update \0014 79;|\012\015");
char            dd15[]          {"|           |          |'ris 57,|\012\015");
char            dd16[]          {"|     N    |\0016 21 | Number of samples used in background time average \0016 79;|\012\015");
char            dd18[]          {"|     P    |\0018 21,| separation of samples used in background time average  |\012\015");
char            dd110[          {"|     d    |\00110 21,| Detection threshold scale factor \00110 79;|\012\015");
char            dd112[]         {"|     n    |\00112 21,| 3 * memory interval length used in alarm decision \00112 79;|\012\015")
char            dd114[]         {"|     t    |\00114 21,| Number of space-time threshold violations to cause alarm|\012\015");
char            dd116[]         {"|     C2   |\00116 21,| Upper threshold used to initiate stare time decrease   |\012\015");
char            dd118[]         {"|     C1   |\00118 21,| Lower threshold used to initiate stare time increase   |\012\015");
char            dd119[]         {"|*\0111*\0111*\0711");
char            disp2[]         {"'ris 17 To change a parameter  type symbol shown above \004");
char            seelst[]        {"'ris 17,To see a list of other features, type  L  ");
char            blnk[]          {"'mon h,'era m,");
char            daterr[]        {"'ris 17,Invalid input command, parameter, or sequence ");
char            dater2[]        {"'ris 17,Invalid feature selection ");
char            sk21[]          {012,015,012,015,0}
char            disp3[]         {"'wor h,'era w,\0011 26,IRCCD INTRUSION DETECTOR ALARM");
char            disp3b[]        {"\0013 31,Additional Features ");
char            work[]          "'wor 28,");
char            learn[]         "'lea et /200006/,");
char            line9[]         {"'ris 17,Please type number of desired feature \004");
char            skip[]          "\012\015'ris 17,");
char            line1a[]        "1) Disable audible alarm ");
char            line1b[]        {"1) Enable audible alarm ");
char            line2[]         "2) Display output of interface A/D converter ");
char            line3[]         {"3) Programable digital shift circuitry test ");
char            line4[]         "4) DMA channel test ");
char            line5[]         "5) RAM integrity test ");
char            line6[]         "6) PROM checksum test ");
char            line7[]         "7) Serial interface/terminal test ");
char            line8[]         {"8) Do nothing - return to parameter display ");
char            home[]          {"'mon h,'ur 10 \015'dow 4,'ris 38,");
char            help1[]         {"'wor h,'um 29 19,NOTE  Calculations are falling behind ");
char            binloc[]        "'wor h,\00110 3,");
char            potloc[]        ("\00119 64.");
char            adtst1[]        ("\0015 28.Outrut of A/D converter \0018 6.Binary\0018 67.Decimal");
char            finloc[]        ("\00119 39.");
char            pdsdis1[]       ("\0015 16.Outrut of Programmible Digital shift circuitry ");
char            pdsdis2[]       ("\0018 6.Binary\0018 37.F1  F2\0018 67.Decimal");
char            test[]          ("test \0015");
char            mkwor[]         ("era m.'era w.'wor 33 h.");
char            mkmon[]         ("era m.'era w.'mon 34 h k.");
char            prmdis1[]       ("\0015 32.PROM Checksums ");
char            prmdis2[]       ("\0018 9.Lower 4K\0018 62.Upper 4K");
char            prmdis3[]       ("\00110 4 Actual   |   Expected\00110 57;Actual   |   Expected\00111 13.|\00111 66;|");
char            ck1loc[]        ("\00111 4.");
char            ck2loc[]        ("\00111 57.");
char            e 1loc[]        ("\00111 19.");
char            e 2loc[]        ("\00111 71.");
char            promok[]        ("\00120 22.No checksum errors detected in PROM ");
char            ckerin1[]       ("\00120 19.Checksum error detected in lower 4K of PROM ");
char            ckerin2[]       ("\00122 19.Checksum error detected in upper 4K of PROM ");
char            ramer[]         ("\00110 20.Error detected in RAM at location ");
char            rtceh[]         ("\00112 6 \005proceed with test, anything else to terminate test ");
char            ramcnt[]        ("\00120 16.Total number of errors detected in RAM was ");
char            ram1[]          ("\0015 31.RAM Integrity Test");
char            noram[]         ("\00120 18.No errors detected during RAM integrity test ");
char            hitret[]        ("\00124 25 \005terminate test ");
char            hitrt2[]        ("dow 3.'rig 23.\005proceed with test ");
char            clscrn[]        ("wor h.\00122 1.");
char            box1[]          ("wor h.\00122 30.");
char            box2[]          ("\00123 30.");
char            box3[]          ("\00124 30.");
char            box4[]          ("\00125 30.");
char            box5[]          ("\00126 30.");
char            star1[]         ("*********************");
char            star2[]         ("*                   *");
char            targ1[]         ("wor h.'rum 24 32.  TARGET DETECTED  'mon h.");
char            notar1[]        ("wor h.'rum 24 32.NO TARGET DETECTED'mon h.");
char            alds1[]         ("wor h.'rum 24 32.  ALARM DISABLED  'mon h.");
char            DMA11[]         ("\0015 36.DMA Test");
char            WCRer[]         ("\0018 19.a)\003R/W of WCR ");
char            noWCR[]         ("\0018 19.a)\002R/W of WCR ");
char            BARer[]         ("\0019 19.b)\003R/W of BAR ");
char            noBAR[]         ("\0019 19.b)\002R/W of BAR ");
char            DBRer[]         ("\00110 19.c)\003R/W of DBR ");
char            noDBR[]         ("\00110 19.c)\002R/W of DBR ");
char            RSTer[]         ("\00111 19.d) Improper response to INIT signal detected ");
char            noRST[]         ("\00111 19.d) Proper response received from INIT signal ");
char            FCTer[]         ("\00112 19.e)\003function-status lines ");
char            noFCT[]         ("\00112 19.e)\002function-status lines ");
char            INTer[]         ("\00113 19.f) End of transfer interrupt not detected ");
char            noINT[]         ("\00113 19.f) End of transfer interrupt detected ");
char            TRNer[]         ("\00114 19.g)\003data transfer test ");
char            noTRN[]         ("\00114 19.g)\002data transfer test ");
char            atod1[]         ("\0015 25 A/D Converter Test Description ");
char            atod2[]         ("\0017 10.1) Twelve bit A/D converter output will be displayed on");
char            atod3[]         ("\0018 12.screen in both binary and decimal ");
char            pds1[]          ("\0014 12.Programmable Digital Shift Circuitry Test Description.");
char            pds2[]          ("\0016 10.1) Output of programmable digital shift circuitry will be displayed");
char            pds3[]          ("\0017 13.on screen in both binary and decimal ");
char            pds4[]          ("\0019 10.2) Function 1 and 2 lines will step through all possible");
char            pds5[]          ("\00110 13.combinations ");
char            pds6[]          ("\00112 10.3) Number of bits displayed are related as follows: ");
char          pds7[]        "\00114 34,1  F2 1 No. of bits"),
char          pds8[]        "\00115 34,1   1 1      9"),
char          pds9[]        "\00116 34,0   1 1     10"),
char          pds10[]       "\00117 34,1   0 1     11"),
char          pds11[]       "\00118 34,0   0 1     12"),
char          dma1[]        "\0015 25.DMA Channel Test Description "),
char          dma2[]        "\0017 10.1) NOTE - DMA MAINTENANCE CABLE MUST BE IN PLACE BEFORE"),
char          dma3[]        "\0018 13.PERFORMING THIS TEST "),
char          dma4[]        "\00110 10.2) Various aspects of DMA operation are tested and results"),
char          dma5[]        "\00111 13.displayed for each seperate test "),
char          dma6[]        {"\00114 14.After halting   a) Power down "),
char          dma7[]        "\00115 30 b) Connect DMA maintenance cable "),
char          dma8[]        "\00116 30.c) Restart program at beginning "),
char          ramin1[]      "\0015 25.RAM Integrity Test Description "),
char          ramin2[]      {"\0017 10.1) R/W function of each word of RAM is tested "),
char          ramin3[]      "\0018 10.2) Routine halts when error is detected and displays location"),
char          ramin4[]      "\00110 13.(in octal) where error occured "),
char          ramin5[]      {"\00112 10.3) The accumulated error count is displayed in decimal when"),
char          ramin6[]      "\00113 13.the test is terminated ");
char          prom1[]       {"\0015 25.PROM Checksum Test Description "),
char          prom2[]       {"\0017 10.1) PROM checksums are computed and displayed for both"),
char          prom3[]       {"\0018 13.the lower and upper 4K words of PROM "),
char          prom4[]       {"\00110 10.2) Comparison is then made with expected checksums "),
char          ser1[]        "\0014 19.Serial Interface/Terminal Test Description "),
char          ser2[]        {"\0016 10.1) Standard 4024 terminal test is executed. Routine waits"),
char          ser3[]        "\0017 13.for  RETURN  before proceeding "),
char          ser4[]        "\0019 10.2) Monitor and serial interface are tested by displaying");
char          ser5[]        {"\00110 13.continuously changing pattern of ascii characters "),
char          ser6[]        "\00112 14.NOTE - \005terminate this segment of test "),
char          ser7[]        "\00114 10.3) Workspace and serial interface are tested by displaying"),
char          ser8[]        "\00115 13.continuously changing pattern of ascii characters. Display"),
char          ser9[]        "\00116 13.will stop scrolling when internal memory of 4024 is full ");
char          ser10[]       "\00118 13.NOTE - \005terminate test "),
char          hittst[]      {"!mon h.!rig 13.\005proceed or any other key to skip test \004"),
char          sum[]         "  sum "),
char          Noer[]        {" No error detected in ");
char          Err[]         {" Error detected in "),
char          hithlt[]      {"!mon h.!rig 13.\005halt or any other key to skip test \004");
char          hitrep[]      {"\00124 25.\005repeat test "),
char          hitstr[]      {"\00124 22.\005restart computations ");
char          hitrt[]       {"Hit  RETURN  to "),

/* spcer is a dummy array used to ensure PROM checksums are stored in
   the correct locations */

int           spcer[]       {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0,0};

/* CKSM1 and CKSM2 contain the correct checksums for the lower
   and upper 4K words of PROM respectively */

int           CKSM1         {061224,
 161          CKSM2          0174240,
/*
*****************************************************************
**                                                             **
**  IRCCD INTRUSION DETECTOR ALARM -                           **
**  This is the terminal handling routine for the IRCCD intrusion**
**  detector alarm. The basic philosophy for I/O in this program **
**  is that data to and from the terminal will be sent under    **
**  program control versus being interrupt driven. This allows  **
**  the computational routine to have complete control of the   **
**  processor whenever it needs it, thus forcing terminal handling**
**  functions to be performed only when nothing is being done.  **
**  Also contained in this section of code are self-diagnostic  **
**  routines to aid in trouble-shooting the detector should the **
**  need ever arise                                             **
**                                                             **
*****************************************************************
*/



/* include variable definitions file for use in compiling */
#include "deffirccd.h"



/* MAIN - routine to control basic program flow
   This is the section of code that controls all terminal handling
   functions. Character data is read from or written to the
   Tektronixs 4024 terminal under control of this routine
*/
main() {
/* go to initialization routine    */
initial();
start:
/* test for incoming character at serial interface   */
if(*stinstat==0) goto write;
/* get character from buffer and take appropriate action    */
ttyin();
goto start;
write:
/* test if there is anything to write out to the tty    */
if(bfptl == bfptotl && bfptl2 == bfptot2) goto start;
/* send characters to tty    */
send();
goto start;



/* TTYIN - incoming character handling routine
   This routine takes data that has been read from the
   terminal and calls routines which in turn decode this
   data into commands or parameter values as neccessary
   Additional routines are then called which take the
   appropriate action to service these commands
*/
ttyin() {
char c,*d;
int i;
```

```
d=TTIBUF;          /* set pointer to serial interface input buffer */
c= *d & 0177; /* remove any parity bits from incoming character    */
if(c == 0100) { /* delete line  (@,ctrl-x,or del line keys) */
   puttty1(home);        /* position cursor  */
   puttty1(blank);       /* clear line on screen */
   tticnt=0; /* reset buffer count to 0 */
   ttiptrib=ttibuf; /* reset pointer to beginning of input buffer */
   return;
}
if(c == 043) {  /* delete character (#,ctrl-h, or del char keys)  */
   if(tticnt) {          /* make sure not at start of line */
      tticnt--;          /* decrement input buffer counter */
      puttty1(bksp);     /* put backspace on output stack */
      --ttiptrib;        /* move pointer to input buffer back one char */
   }
   return;
}
if(c != 015) {           /* check if char is not CR (eol) */
   if(tticnt==9) return;          /* check if input buffer is full */
   *ttiptrib++=c;        /* put char in buffer and increment pointer */
   tticnt++;             /* increment input buffer counter */
   holdchar[0]=c;        /* put char in temp location to allow placement on stack */
   holdchar[1]=0;        /* add on eol char */
   puttty1(holdchar);    /* put char on output stack */
   return;
}
*(ttiptrib++)=c;         /* put last char of line in input buffer */
tticnt=0;        /* set input buffer count back to zero */
puttty1(crlf);  /* put CR on output stack */
send(); /* empty output buffers before executing command */
/* parse and decode command line    */
parse();
command=decode();        /*  command  contains decoded command number */
if(command==0 && o.dcomnd!=9) {  /* invalid command,parameter,or combination*/
      bds();     /* put error message on output stack */
      puttty1(disp2)     /* put refresh message on screen */
      puttty1(home);     /* reposition cursor */
}
if(command==0 && o.dcomnd==9) { /* check for invalid feature selection */
   puttty1(blnk);        /* clear bottom of screen */
   puttty1(dater2)       /* put up error message */
   puttty1(sl2);         /* skip 2 lines    */
   puttty1(line9)        /* refresh  which feature  line */
}
/* now branch to proper command execution routine */
switch(command) {

case 1  Ddist();         /* list possible values of D parameter */
        break;
case 2  Nlist()          /* list possible values of N parameter */
        break;
case 3  Plist();         /* list possible values of P parameter */
        break
case 4  dlist();         /* list possible values of d parameter */
        break;
case 5  nlist();         /* list possible values of n parameter */
        break;
case 6  tlist()          /* list possible values of t parameter */
        break;
case 7  c2list();        /* list possible values of C2 parameter */
```

```
        break;
case 8  c1list();       /* list possible values of C1 parameter */
        break;
case 9  addfeat();      /* put up list of additional features */
        break
/* If valid command before, go to routine which services that command */
case 10  switch(oldcomnd) {

case 1  ddsru();        /* go to change D parameter */
        break;
case 2  nnsru()         /* go to change N parameter */
        break;
case 3  ppsru();        /* go to change P parameter */
        break;
case 4  dsru();  /* go to change d parameter */
        break;
case 5  nsru()  /* go to change n parameter */
        break;
case 6  [illegible]   /* go to change [illegible] parameter */
        break
case 7  c2sru()         /* go to change C2 parameter */
        break
case 8  c1sru()         /* go to change C1 parameter */
        break
case 9  lsru(); /* go to routine which services additional features list */
        break;
case 10  bds(); /* put up error message */
         puttyl(disp2);
         puttyl(home)
        break;
}
break;
case 11  if(oldcomnd==10) {     /* check if line only consisted of CR */
         /* put up additional features list    */
            addfeat();
         }
         else {
               if oldcomnd==9) {       /* put up initial parameter display */
                  putdsp();
               }
            else {
               bs( );    /* put up error message */
               puttyl(disp2)
               puttyl home)
        oldcomnd=1

         }
         break;
}
/* reset pointers etc    */
[illegible]=[illegible]buf
return
}



/* PARSE - routine for parsing command line.
   places command line into the array "string"
*/
parse()
```

- B10 -

```
int j;
ttiptrob=ttibuf;          /* set pointer to beginning of buffer */
if(*ttiptrob==015) {      /* check for CR */
   *string[0]= *ttiptrob;        /* put char into array */
   *(string[0]+1)=0;      /* add eol char */
   return;
}
   for(;*ttiptrob == 040; ttiptrob++);  /* skip spaces */
   if(*ttiptrob == 015) return; /* check for CR */
   for(j=0;(*ttiptrob != 040) && (*ttiptrob != 015);j++) {
      if(j>9) j=9;        /* j points to each element in array */
      *(string[0]+j)= *ttiptrob++;        /* put chars into array */
   }
   *(string[0]+j)=0       /* add eol char */
return;
}



/* DECODE - command decoding routine  Returns command number
   (0 if illegal)
   Places command arguments in proper locations for use
   by command servicing routines
*/
decode()
{
int i,k,m;
k=0     /* k is the returned command number     */
for(i=0; i<13; i++) {
   if(*string[0]== *cmds[i])     /* check which command by comparing to list */
      k=cmdno[i];         /* set k to command number */
      i=13;
   }
}
if(k==7) {        /* check if C1 or C2 command */
   if(*(string[0]+1)== *(cmds[7]+1)) k=8;        /* check if C2 */
   else if(*(string[0]+1)== *(cmds[6]+1)) k=7   /* check if C1 */
   else k=0;     /* set k to error value */
}
if(k==0) {
/* check if input string is numeric data */
   if((*string[0] >= 060 && *string[0] <= 071) || *string[0]==056) k=10;
}
/* decode numeric string in appropriate mannner    */
if(k==10) {
   m=ascton(); /* "ascton" returns mantissa in "datareg" */
   if(m==1) k=0;          /* m indicates whether or not there was an error in conversion */

}
if(oldcomnd==9 && k!=10) k=0;   /* set error if not numeric when expecting */
if(*string[0] == 015) k=11;      /* check if command is CR only (to toggle display,etc.)*/
return(k);
}



/* ASCTON - routine for converting numeric ascii strings into
   appropriately formatted binary numbers
*/
ascton(){
```

```
int i,j,k,n,m,dflg,erflg,lim;
int frac,pow,carry;
char c;
dflg=erflg=0;   /* initialize some internal flags */
n=m=0;  /* initialize number of integer and fractional places */
/* determine the number of integer places (n) and the number
   of decimal places (m)     */
for(i=0;c= *(string[0]+i);i++) {         /* go till end of line */
   if(c!=056 && dflg==0) n++    /* count number of decimal places before decimal point */
   if(dflg==1) m++;     /* count fractional places */
   if((c<060 && c!=056)||(c>071)||(c==056 && dflg==1)) erflg=1; /* check for error */
   if(c==056) dflg=1;   /* check for decimal points */
}
if(m>2) m=3;    /* allow for rounding of floating point numbers */
if(oldcomnd!=4 && m>0) m=1;     /* allow for rounding of floating point values to integers */
if(m>2 || (oldcomnd!=4 && m>0)) {       /* round of floating point entries */
   if(string[0][n+m]>=065) {
      /* round m decimal places to two places    */
      carry=1
      for(j=m-1;carry!=0 && j>0;j--) {  /* go till carry not equal to zero */
         string[0][n+j]=+ carry;        /* add in carry from previous place */
         if(string[0][n+j]>071) string[0][n+j]=060;/* wrap around after carry if appropriate */
         else carry=0;
      }
      for(j=n-1;carry!=0 && j>=0;j--) { /* round integer part */
         string[0][j]=+ carry;
         if(string[0][j]>071) string[0][j]=060;
         else carry=0;
      }
      if(carry==1) {    /* add leading 1 if carry out of last place */
         string[0][0]=061;
         for(j=1;j<=n+3;j++) string[0][j]=060   /* set remaining places to zero */
         string[0][n+1]=056      /* add decimal point */
         n++;
      }
   }
}
if(m<2) {        /* round to integer values when not changing d parameter */
   string[0][n]=056;
   for(j=n+m+1;j<n+3;j++) string[0][j]=060;
}
m=2     /* set to 2 fractional places */
if(n>3) n=3;    /* truncate to 3 integer places */
if(oldcomnd!=4) lim=n;  /* check if not changing d parameter (only have n integer places) */
else lim=n+m+1  /* set total number of places for changing d value */
for(j=0;j<lim;j++) datstrng[j]=string[0][j];/* copy string into buffer for displaying on screen */
datstrng[j]=0;  /* add eol char */
if(erflg==1) return(1); /* set error flag */
/* convert integer part of data from ascii to binary */
datares=0;
/* add in weighted value of each digit */
for(i=0;i<n;i++) datares=datares*10+(*(string[0]+i)-060);
/* convert fractional data    */
if(oldcomnd!=4) return(0);
else {
frac=0;
for(i=0;i<m;i++)
/* add in weighted value of fractional digits   */
   frac=frac*10+(*(string[0]+n+1+i)-060);
frac=* 10;
```

```
powr=1000;
for(i=0;i<8;i++) {
   frac=frac*10;
   if(frac>=powr) {      /* check which bits to set in datares */
      datares=(datares<<1) | 01;          /* set these bits */
      frac-= powr;
   }
   else datares=datares<<1;      /* don't set bit in datares */
}
datares++;       /* round off fractional bits   */
datares=datares>>1;
/* datares is in binary format dddddddd.dddddd when decoding
   d values and is a normal integer for all other parameters */
return(0);
}
}



/* SEND - routine for sending characters to terminal via
   circular output queue (outstack) "bufptin" and "bufptout"
   are the input and output pointers to this queue.
   Two queues are actually used-one for information from the
   user controlled functions and one for information from
   the interrupt controlled functions
*/
send() {
int cntr,i,j,k,bytecnt,*stackpt;
char c,*d,**ptin,**ptout;
d=TTOBUF;
ttostat=TTYOSTAT;
for(i=1;i>=0;i--) {
   if(i==0) {              /* determine which stack is being sent out */
      ptout=bfptot2;               /* set output pointer to interrupt stack */
      ptin=bfpti2;                 /* set input pointer to interrupt stack */
      bytecnt=btcnt2;              /* bytecnt tells which byte in the current ascii string is being sent to tty */
      stackpt=outstk2;  /* use interrupt output queue */
   }
   else {
      ptout=bfptot1;               /* set output pointer to user stack */
      ptin=bfpti1;                 /* set input pointer to user stack location */
      bytecnt=btcnt1;   /* set which byte in string to start with */
      stackpt=outstk1;  /* use user output queue */
   }
   cntr=0;
   while(ptin != ptout) {        /* test if stack empty    */
      while(*ttostat>=0)          /* test status of ready bit on serial interface */
      c= *(*ptout+bytecnt);       /* get next character    */
      if(i==0) {                  /* service interrupt stack    */
         if(c==33) cntr=1;        /* check for command character(!)... */
         else if(c==59 && cntr!=0) {
            cntr=0;
            *d=c;                 /* send out ; character    */
 /* send out some nulls to let term catch up    */
            for(j=0;j<10;j++) {
                while(*ttostat>=0);
                *d=0;
            }
            j=0;
            while(*ttostat>=0);
            }
        }
        else {
            if(c==01) {              /* send out 'rum' sequence ... */
                for(j=0; j<5; j++) {
                    *d=rum[j];
                    while(*ttostat!=0);
                }
                c=0;
            }
            if(c==02) {              /* send ' No error detected in ' */
                for(j=0; j<22; j++) {
                    *d=Noer[j];
                    while(*ttostat!=0);
                }
                c=0;
            }
            if(c==03) {              /* send ' Error detected in ' */
                for(j=0; j<19; j++) {
                    *d=Err[j];
                    while(*ttostat!=0);
                }
                c=0;
            }
            if(c==04) {              /* send 'home' sequence   */
                for(j=0; j<30; j++) {
                    *d=home[j];
                    while(*ttostat!=0);
                }
                c=0;                 /* send null char    */
            }
            if(c==05) {              /* send "Hit 'RETURN' " sequence .  */
                    for(j=0; j<16; j++) {
                        *d= hitrt[j];
                        while(*ttostat!=0);
                    }
                    c=0;
            }
            if(c==0137) {            /* check for underscore repeat    */
                k= *(*ptout+(++bytecnt));    /* check how man.    */
                for(j=0; j<k; j++) {
                    *d=0137;
                    while(*ttostat!=0);
                }
                c=0;
            }
        }
        *d=c;
        bytecnt++           /* go to next char in string */
        c= *(*ptout+bytecnt);
        if(c==0) {          /* check for end of line char     */
            bytecnt=0;
            ptout++;
            if(ptout>stacket+40) ptout=stacket;     /* wrap around circular queue */
        }
    }
    if(i==0) {              /* reset pointers    */
        bfptot2=ptout;
        btcnt2=bytecnt;
    }
```

```
   else {
      bfptl1=ptout.
      btcnt1=bytecnt.
   }
}
return.
}



/* PUTTTY1 - routine to place location of outgoing ascii on
   output queue list no. 1
   This is information from user controlled functions
*/
puttty1(loc)
int loc. {
*bfptl1=loc.     /* set pointer to start of string */
bfptl1++.
if(bfptl1>(outstk1+40)) bfptl1=outstk1. /* check for wrap around on queue */
}



/* PUTTTY2 - routine for placing location of outgoing ascii from
   interrupt service routines on output queue list no. 2
*/
puttty2(loc)
int loc. {
*bfptl2=loc.     /* set pointer to start of string */
bfptl2++.
if(bfptl2>(outstk2+40)) bfptl2=outstk2. /* check for wrap around on queue */
}



/* INITIAL - initialization routine for IRCCD intrusion
   detector. Called at turn on time and when routine is
   restarted after performing diagnostic tests
*/
initial() {
int *d;
extern int dummy.
/* disable interupts temporarily   */
d=CSR.
*d=& 0177667.
d=INTVEC.
/* place starting address of dummy routine in DMA interrupt
   vector location */
*d++= &dummy.
*d=040.
d=WCR
*d= 0177400      /* set DMA WCR to initial value (-256) */
d=BAR.
*d= xbuf1.       /* set DMA BAR pointing to xbuf1 for first packet*/
stinstat=TTYSTAT..       /* set pointer to serial int. input status byte*/
oldcomnd=10.
aldisab=0.
flag=0.
ttyfrit=ttibuf.         /* set pointer to input buffer */
```

```
[illegible]=[illegible]          /* set pointer to output buffer */
[illegible]=0         /* set buffer count to 0 */
/* set pointers to various output queues */
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]=0
[illegible]
[illegible]=0          /* set initial char cnts to 0 */
[illegible]=0        /* flag to tell when calculations should be disabled */
[illegible]dalrm=0,      /* turn on audible alarm initially */
bptr= &htab      /* set pointer to start of table used in computations */
aptr= &a        /* set pointer to start of a vector */
pcnt=0. /* counter to tell which pth of computations currently on*/
[illegible]=0
[illegible]=6  /* set f1 f2 initially to 1s */
[illegible]=6.
oldtre[illegible]=6
bigm=32         /* initial value of N parameter */
bigp=12,        /* initial value of P parameter */
bufnum=0.       /* start with sbuf1 for first packet */
/* initialize tek 4025 terminal    */
puttty1(work).  /* define workspace etc .  */
puttty1(learn). /* program PI key to go sequence */
/* program various erase char and erase line keys */
puttty1(f9).
puttty1(f10).
puttty1(ctrh).
puttty1(ctrx).
/* Test if DMA test cable is in place   */
d=CSR.
tmp1= (~((*d & 070000)>>8)) & 016.
*d=& 0177761    /* set function lines to 0 */
*d=| tmp1
tmp2= (*d & 070000)>>8
if(tmp1 == tmp2) dmatst().        /* check if status = function lines */

/* put up initial display    */
display1().
[illegible]
puttty1(disp2)
[illegible]
alarm()
puttty1(home)
[illegible] /* empty queues to terminal */
/* set default parameter values used in calculations   */
/* D value = 15 */
datareg=15
datstrng[0]=061
datstrng[1]=[illegible]
datstrng[2]=0
ddrout()
/* N value = 32 */
datareg=32
datstrng[0]=063.
datstrng[1]=062
datstrng[2]=0.
nnsrout()
/* P value = 12 */
```

```
datareg=12.
datstrng[0]=061.
datstrng[1]=062.
datstrng[2]=0.
essro()
/* n value = 12 */
nsro();
/* C2 value = 480 */
datareg=480
datstrng[0]=064.
datstrng[1]=070.
datstrng[2]=060.
datstrng[3]=0.
c2sro().
send(). /* write out output queues */
/* C1 value = 230 */
datareg=230.
datstrng[0]=062.
datstrng[1]=063.
datstrng[2]=060
datstrng[3]=0.
c1sro()
/* t value = 8 */
datareg=8.
datstrng[0]=070
datstrng[1]=0
tsro()
/* d value = o 17 */
/* format of d is nnnn nnnnnnn */
datareg=0142e.
datstrng[0]=06e
datstrng[1]=05e.
datstrng[2]=061.
datstrng[3]=067.
datstrng[5]=0.
dsro().
send().
*DCSR
*d=[illegible]0100          /* enable interrupts from DMA */
/* bit no bit        */
/* (will wait for a few microseconds to make sure interupt has hit) */
/* set alarm disable count */
alditab=bias*bias+1.     /* this should do it     */
*d=1 01
/* raise function three line     */
*d=1 010.
return.
}



/* DDLIST - routine to list "D" values   */
ddlist() {
bss().
puttty1(ddlist).
oldcomnd=1.
}

/* NNLIST - routine to list "N" values    */
nnlist() {
```

```
bss();
puttty1(nlist);
oldcomnd=2;
}

/* PPLIST - routine to list "P" values   */
pplist() {
bss();
puttty1(pplist);
oldcomnd=3;
}

/* DLIST - routine to list "d" values    */
dlist() {
bss();
puttty1(dlist);
oldcomnd=4;
}

/* NLIST - routine to list "n" values    */
nlist() {
bss();
puttty1(nlist);
oldcomnd=5;
}

/* TLIST - routine to list "t" values    */
tlist() {
bss();
puttty1(tlist);
oldcomnd=6;
}

/* C2LIST - routine to list "c2" values   */
c2list() {
bss();
puttty1(c2list);
oldcomnd=7;
}

/* C1LIST - routine to list "c1" values   */
c1list() {
bss();
puttty1(c1list);
oldcomnd=8;
}

/* DCHG - routine to change "D" parameter    */
dchg() {
int i;
/* check for allowed value of D parameter */
if(datares!=1 && datares!=2) {
   bds();          /* put up error message */
   puttty1(ddlist);
   return;
}
/* temporarily disable calculation routines */
intflg=1;
bigd=datares;    /* set new D value */
/* put ascii string of parameter value in buffer */
```

```
for(i=0; datstrng[i]; i++) bigdstr[i]=datstrng[i];
bigdstr[i]=0;
ntbkspan=256/bigd;        /* set number of terms used in computing background spat avg */
tbgd=bigd<<1;    /* 2*D value used in calculation routine */
/* set number of shifts to divide by D */
if(bigd==16) dshft= -4;
else dshft= -5;
/* enable calculations */
intflg=0;
puttty1(ddloc);
puttty1(bigdstr);         /* send new value of D to terminal */
/* refresh bottom of screen */
bss();
puttty1(disp2);
oldcomnd=10;
return;
}

/* NNGRV - routine to change "N" parameter    */
nngrv() {
int i;
/* check for valid N parameter values */
if(datareg<2 || datareg>40) {
   bdz();         /* put up error message on screen */
   puttty1(errlst);
   return;
}
/* temp. disable calculation routine */
intflg=1;
bign=datareg;    /* set new value of N */
/* look up sqrt(N+1/N) in table */
sqn=npl[bign-1];
thresh();         /* recompute new threshold table used in computations */
for(i=0; datstrng[i]; i++) bignstr[i]=datstrng[i];
bignstr[i]=0;
wrap=512*bign;   /* value to tell when vector holding past data is full */
pcnt=0;  /* counter to tell which pth of computations on */
rdtrsh=(bign+1)>>1;      /* value used to aid in rounding off operation */
xptr= &x;         /* set pointer to start of x vector */
mptr= &m;         /* set pointer to start of m vector */
for(i=0; i<(bign<<8); i++) x[i]=0;         /* clear x vector */
for(i=0; i<1024; i++) m[i]=0;      /* clear m vector */
yincr=0;          /* clear pointer to element in ybuf */
np=bign*bigp;    /* compute N*P */
aldisab=np;      /* set alarm disable counter */
/* enable computations */
intflg=0;
/* send new value to tty */
puttty1(nnloc);
puttty1(bignstr);
/* refresh bottom of screen */
bss();
puttty1(disp2);
oldcomnd=10;
return;
}

/* PPGRV - routine to change "P" parameter    */
ppgrv() {
int i;
```

```
/* check if valid P parameter value */
if(datareg<10 || datareg>256) {
   bds();        /* send error message to tty */
   puttty1(pelst);
   return;
}
/* temporarily disable computations */
intflg=1
bige=datareg    /* set new value of P */
for(i=0;datstrng[i];i++) bigestr[i]=datstrng[i];
bigestr[i]=0;
/* compute how computational load should be divided   */
/* enorm tells how many computations will be done the first
   nl-1 iterations while plast tells how many will be done the
   nlth time nl may or may not be equal to P depending on the
   current value of P .
*/
nl=bige-1;
pcnt=0;
enorm=((512/bige)+1)>>1;
tenorm=enorm<<1;
plast=256-enorm*nl;
if(plast>25) {  /* allow maximum of 25 computations each time */
        enorm++;
        tenorm=enorm<<1;
        plast=256-enorm*nl;
}
while(plast<=0) {
        nl--
        plast=256-enorm*nl;
}
nl++;
xptr= &x        /* set pointer to x vector */
for(i=0;i<(tiar<<8);i++) x[i]=0;         /* clear x vector */
mptr= &m        /* set pointer to start of m vector */
/* clear m vector */
for(i=0;i<1024;i++) m[i]=0
/* set pointer to starting location in buf */
 inptr=0
nr=biane*bias    /* compute N*P */
aldisat=nr       /* set alarm disable to N*P */
/* enable calculations */
intflg=0;
/* send new P value to tty */
puttty1(pel);
puttty1(bigestr);
bos();   /* refresh bottom of screen */
puttty1(dirsl);
oldcomnd=10;
return;
}

/* DISV - routine to change "d" parameter.   */
dsmv()
int i;
/* check for valid d parameter values */
if(datareg<=0 || datareg>=02400) {
   bds();        /* send error message to tty */
   puttty1(dlst);
   return;
```

```
smd=datares;     /* set new value of d parameter */
for(i=0;datstrng[i];i++) smdstr[i]=datstrng[i];
smdstr[i]=0;
thresh();        /* compute new threshold table for use in computations*/
/* send new d value to tty */
puttty1(dloc);
puttty1(smdstr);
bss();
puttty1(disp2);
oldcomnd=10;
return;
}

/* NSRV - routine to change "n" parameter   */
nsrv() {
int i;
/* check for valid n parameter values */
if((datares!=6 && datares!=9 && datares!=12 && datares!=15) || datares<smt) {
   bds();        /* put error message on tty */
   puttty1(nlst);
   return;
}
/* temporarily disable computations */
intflg=1;
smn=datares;     /* set new n value */
for(i=0;datstrng[i];i++) smnstr[i]=datstrng[i];
smnstr[i]=0;
mask=(017770000>>(5-smn/3)) | 01;   /* set up mask for comp routine*/
if(aldisab<smn/3) aldisab=smn/3;        /* set alarm disable */
/* enable computations */
intflg=0;
/* send new n value to tty */
puttty1(nloc);
puttty1(smnstr);
bss();
puttty1(disp2);
oldcomnd=10;
return;
}

/* TSRV - routine to change "t" parameter   */
tsrv() {
int i;
/* check for valid t parameter values */
if(datares>smn || datares<3 || datares>15) {
   bds();        /* send error message to tty */
   puttty1(tlst);
   return;
}
/* set new t value */
smt=datares;
for(i=0;datstrng[i];i++) smtstr[i]=datstrng[i];
smtstr[i]=0;
/* send new t value to tty */
puttty1(tloc);
puttty1(smtstr);
bss();
puttty1(disp2);
oldcomnd=10;
```

```
return;
}

/* C2SRV - routine for changing "C2" parameter.  */
c2srv() {
int i;
/* check for valid C2 parameter values */
if(datares!=(c1<<1) || datares>511) {
   bds();          /* send error message to tty */
   puttty1(c21st);
   return;
}
/* set new C2 parameter value */
c2=datares;
for(i=0;datstrng[i];i++) c2str[i]=datstrng[i];
c2str[i]=0;
/* temp. disable computations */
intflg=1;
/* compute multiples of C2 for use in stare time changes */
cc2[3]=c2;
for(i=2;i>=0;i--) cc2[i]=cc2[i+1]<<1;
/* enable computations */
intflg=0;
/* send new C2 vallue to tty */
puttty1(c2loc);
puttty1(c2str);
bss();
puttty1(disp2);
oldcomnd=10;
return;
}

/* C1SRV - routine for changing "C1" parameter   */
c1srv() {
int i;
/* check if valid C1 parameter value */
if(datares>=((c2+1)>>1) || datares<0) {
   bds();          /* send error message to tty */
   puttty1(c11st);
   return;
}
/* set new value of C1 parameter */
c1=datares;
for(i=0;datstrng[i];i++) c1str[i]=datstrng[i];
c1str[i]=0;
/* temp. disable computa[illegible] */
intflg=1;
/* compute multiple[illegible] stare time changes */
cc1[3]=c1;
for(i=2;i>=0;i--) cc1[i]=cc1[i+1]<<1;
/* enable computations */
intflg=0;
/* send new C1 value to tty */
puttty1(c1loc);
puttty1(c1str);
bss();
puttty1(disp2);
oldcomnd=10;
return;
}
```

```
/* LSRV - routine to do "other features"    */
lsrv() {
int c,*d;
d=TTIBUF;
switch(datares) {
/* change status of audible alarm */
case 1: audalrm=(audalrm+1)&01;
        break;
/* go to routine which tests the A to D converter */
case 2: atodtst();
        return;
/* go to routine which tests the programmable digital shift logic */
case 3: pdstst();
        return;
/* go to routine which displays the DMA test instructions */
case 4: dmains();               /* put up DMA test instructions   */
        while(*stinstat==0) send();     /* wait for halt or skip test*/
        c= *d & 0177;
        if(c != 015) addfeat(); /* put up additional feat. list*/
        else halt();    /* execute halt instruction */
        return;
/* go to routine which tests RAM part of memory */
case 5: ramtst();
        return;
/* go to routine which tests PROM part of memory */
case 6: promtst();
        return;
/* go to routine which tests the tty and serial interface */
case 7: termtst();
        return;
}
/* go back to the original display */
putdisp();
}



/* ADDFEAT - routine to put up display of additional features list.
   This list tells the operator which tests he may or may not
   choose to perform
*/
addfeat() {
puttty1(disp3);
puttty1(disp3b);
puttty1(crlf);
puttty1(skip);
/* decide whether to put up audible alarm enable or disable */
if(audalrm==0) puttty1(line1a);
else puttty1(line1b);
puttty1(skip);
puttty1(line2);
puttty1(skip);
puttty1(line3);
puttty1(skip);
puttty1(line4);
puttty1(skip);
puttty1(line5);
puttty1(skip);
puttty1(line6);
puttty1(skip);
putttyl(line7);
putttyl(skip);
putttyl(line8);
putttyl(blnk);
putttyl(sk21)
putttyl(line9)
box();  /* draw box for target messages */
alarm()         /* fill up box with appropriate message */
oldcomnd=9;
}



/* PUTDSP1 - routine for putting initial parameter display list on the
   screen  This is the display that is first seen at turn on time
*/
putdspl() {
display1();     /* draw table at top of screen */
send(); /* empty output queues */
/* fill in various parameter values */
putttyl(ddloc);
putttyl(bigdstr);
putttyl(nnloc);
putttyl(bignstr);
putttyl(pploc)
putttyl(bigpstr);
putttyl(dloc);
putttyl(smdstr);
putttyl(nloc);
putttyl(smnstr);
putttyl(tloc);
putttyl(smtstr)
putttyl(c2loc);
putttyl(c2str)
putttyl(c1loc);
putttyl(c1str);
/* fill in bottom of screen */
bsc();
putttyl(disp2)
box();  /* draw box for target messages */
alarm();        /* fill up box with appropriate message */
oldcomnd=10;
}



/* DISPLAY1 - routine to aid the PUTDSP1 display routine in
   constructing the initial display   This routine actually
   draws the parameter table at the top of the 4024 display
*/
display1();
putttyl(ddl1);
putttyl(ddl2);
putttyl(ddl3);
putttyl(ddl4);
putttyl(ddl5);
putttyl(ddl6)
putttyl(ddl5);
putttyl(ddl9)
```

```
puttty1(ddl5);
puttty1(ddl10);
puttty1(ddl5);
puttty1(ddl12);
puttty1(ddl5);
puttty1(ddl14);
puttty1(ddl5);
puttty1(ddl16);
puttty1(ddl5);
puttty1(ddl18);
puttty1(ddl19);
send(); /* empty output queues */
}



/* ATODTST - routine to test A to D converter
   This routine tests the A to D converter on the custom interface
   board by loading samples into memory via the DMA. The function
   lines of the DMA are set to 1s to allow all 12 bits of the A to D
   to be displayed. It is assumed that start convert and start
   packet signals are being applied to the interface board. These
   allow clocking of the flip-flops for the function lines and provide
   cycle request pulses for the DMA.
*/
atodtst() {
int *b;
char c,*d;
d=TTIBUF;
atodins();      /* put up instructions for A to D test   */
/* Wait for continue symbol (return) or skip test (anything else)*/
while(*stinstat>=0) send();      /* refresh target message while waiting for char*/
c= *d & 0177;   /* remove parity bits, if any     */
if(c != 015) {  /* check if not CR */
   addfeat();   /* put up additional features list */
   return;
}
/* perform A to D test   */
/* disable interupts     */
b=CSR;
*b=& 0177677;
/* set function lines to display all 12 bits of A/D.   */
*b=& 0177771;
/* display A to D output on screen in binary and decimal    */
puttty1(ddl1);  /* IRCCD line at top of screen */
puttty1(adtst); /* test label */
puttty1(bitret);
puttty1(home);
/* repeatedly call routine which gets sample from A to D */
testit: while(*stinstat>=0) atod(); /* wait for some kind of break */
c= *d;
/* terminate test and reset program.   */
initial();      /* call initialization routine */
}



/* ATOD - routine to actually read and display sample from A to D
   This routine is called by both the A to D test and the PDS test
```

```
    For the A to D test, the function lines are set to display all
    12 bits before calling while for the the PDS test, the function
    lines are cycled through there various values
*/
atod() {
char *b;
int *d,i,divide,shift;
/* set up DMA to load in 1 word from interface ...   */
d=WCR;
*d=0177777;
d=BAR;
*d= &atodbyt;
d=CSR;
*d=I 01   /* hit 'go' bit      */
/* test if DMA done loading in word    */
b=CSR;
while(*b`=0)
/* display 'atodbyt' on screen    */
/* convert word from binary to ascii representation of binary */
/* only 12 bits of the 16 bit word are displayed     */
shift=atodbyt;
atdbin[12]=0;
for(i=11;i>=0;i--) {
   atdbin[i]=(shift & 01) + 060;
   shift=shift>>1;
}
/* convert binary word to ascii representation of decimal value */
atodec(atodbyt,atdoct);
/* display binary and decimal versions of word on screen */
puttty1(binloc);
puttty1(atdbin);
puttty1(octloc);
puttty1(atdoct);
/* clear output buffers    */
send();
for(i=0;i<4000;i++)     /* delay to allow terminal sync */
}



/* ATODINS - routine to display instruction frame for A to D test
   A brief description of the A to D test is displayed on the
   screen
*/
atodins() {
puttty1(ddl1);
puttty1(atod1);
puttty1(atod2);
puttty1(atod3);
puttty1(hittst);
box();   /* draw box for target messages */
alarm();         /* fill box with appropriate message */
send(); /* empty output queue; */
}



/* PDSTST - routine to test the programmible digital shift logic
   This routine tests the PDS logic circuitry by displaying its
   output on the screen via a DMA transfer  The function lines
   are cycled through the various possible combinations and the
   appropriately shifted version of the output of the A to D are
   displayed on the screen in both binary and decimal. This routine
   is very similar to the A to D test routine except for the feature
   of cycling through the function line values.
*/
pdstst() {
int *b,fln,i;
char c,*d;
d=TTIBUF;
pdsins();     /* put up pds instructions.... */
/* wait for continue symbol (return) or skip test (anything else) */
while(*stinstat>=0) send();     /* empty output queues to display target */
c= *d & 0177;
if(c != 015) {  /* check if not CR */
   addfeat();   /* put up additional features list */
   return;
}
/* perform pds test    */
/* disable interupts   */
b=CSR;
*b=& 0177677;
/* set function lines initially to tref    */
fln=06;
*b=| fln;
/* label screen */
putttyl(ddl1);
putttyl(pdsdis1);
putttyl(pdsdis2);
putttyl(hitret);
putttyl(home);
i=0;
/* initialize buffer used to display value of function lines on tty*/
flines[1]=flines[2]=flines[3]=040;
flines[5]=0;
testit: atod(); /* actually go to routine which displays output of PDS */
/* display function lines    */
flines[0]= ((fln & 02)>>1)+060;
flines[4]= ((fln & 04)>>2)+060;
putttyl(flnloc);
putttyl(flines);
i++;
if(i>20) {   /* delay in changing f lines    */
   i=0;
   fln=+ 2;      /* change function lines */
   if(fln>6) fln=0;      /* check to see if valid combination */
/* send out new function lines */
   *b=& 0177771;
   *b=| fln;
}
if(*stinstat>=0) goto testit;   /* look for break signal */
c= *d;
/* terminate test and reset program    */
[illegible]         /* go to initialization routine */
}



/* PDSINS - routine to display PDS test instructions
   This routine lists a brief description of the test of
   the Programmible Digital Shift logic circuitry
*
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible] 4
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible] draw box for displaying target info */
[illegible]        /* fill in box with appropriate message */
[illegible]  /* empty output queue */



/* TERMTST - routine to test the 4024 terminal and serial interface
   This routine tests the 4024 terminal and serial interface to the
   PDP 11-03 in three stages (1) By performing the standard 4024
   terminal test routine (2) By sending a unique character pattern
   to the monitor area of the terminal (3) By sending a unique
   character pattern to the workspace area of the terminal
*/
termtst() {
int *b,i,k;
char c,*d;
d=TTIBUF;
termins(); /* put up terminal test instructions */
/* wait for continue symbol (return) or skip test (any other) */
while(*stinstat>=0) send();     /* refresh target messages while waiting */
c= *d & 0177;
if(c != 015) {  /* check if not CR */
   addfeat();   /* put up additional features list */
   return;
}
/* perform actual int/term tests    */
/* disable interupts */
b=CSR;
*b=& 0177677;
/* test terminal with standard 4024/4025 test */
putt(1,"test");
send(); /* empty output queue so test is executed */
/* sit and wait for a while    */
for(k=0;k<5;k++) for(i=0;i<20000;i++);
/* construct an 81 character line for repeated display on screen */
k=052;
for(i=0;i<81;i++) {
   if((i+k) != 0137) [illegible]
   else [illegible]=050;  /* [illegible] 4024 command character [illegible] */
}
[illegible]=0;     /* end of line char */
put[illegible]
send();  /* empty output queue */
```

```
/* wait for proceed (ans keyboard entry) */
while(*stinstat==0);
c= *d;
/* fill monitor screen with 81 character lines    */
putttyl(mkmon); /* make screen all monitor space */
while(*stinstat!=0) {   /* check serial interface status */
   putttyl(runkln);      /* put string on output queue */
   send();       /* empty output queues so string is displayed */
   for(k=0;k<5000;k++)  /* wait a while so terminal can catch up */
}
c= *d;
/* fill workspace with 81 character lines.    */
putttyl(mkwor); /* make screen all workspace (except last line) */
while(*stinstat!=0) {   /* check serial interface status */
   putttyl(runkln);      /* put string on output queue */
   send();       /* empty output queue so string is displayed */
   for(k=0;k<5000;k++);  /* wait a while for terminal to catch up */
}
c= *d;
/* reset routine and return    */
initial();       /* go to initialization routine */
}



/* TERMINS - routine for displaying term/interface test instructions.
   A brief description of the 4024 terminal and serial interface test
   routine is displayed on the screen
*/
termins() {
putttyl(ddl1);
putttyl(ser1);
putttyl(ser2);
putttyl(ser3);
putttyl(ser4);
putttyl(ser5);
putttyl(ser6);
putttyl(ser7);
putttyl(ser8);
putttyl(ser9);
putttyl(ser10);
putttyl(hitts1);
box();  /* draw box for target info */
alarm()          /* fill box with appropriate message */
send(); /* empty output queues */
}



/* PROMTST - routine to verify PROM checksums
   Checksums are computed for all of the lower
   4K words of PROM and most of the upper 4K words of
   PROM. The proper values of both of these checksums are
   stored in two words in the upper 4K section. These
   two words are not used in actually computing the checksums
   but are only used for comparison purposes. Actual and
   expected values of checksums are displayed for both the lower
   and upper 4K word sections
*/
promtst() {
```

```
int *b,excks1,excks2,cks1er,cks2er,i;
char *d,c;
d=TTIBUF;
prmins(); /* put up prom test instructions   */
/* wait for continue symbol(return) or end test(anything else) */
while(*stinstat!=0) send();      /* refresh target messages */
c= *d & 0177;
if(c != 015) {   /* check if not CR */
   addfeat()    /* put up additional features list */
   return;
}
/* perform actual tests
   there will be one checksum for each 4k prom board
*/
/* disable interupts    */
b=CSR;
*b=& 0177677;
/* label screen */
putttyl(ddl1);
putttyl(prmdis1);
putttyl(prmdis2);
putttyl(prmdis3);
send(); /* empty output queues */
/* compute actual checksums   */
b=STL4K          /* set pointer to address of start of lower 4K */
chksm1=0;        /* clear checksum accumulator */
while(b <= ENDLFK ) chksm1=+ *b++;       /* compute lower 4K checksum */
/* b is now left pointing at start of upper 4k */
chksm2=0;        /* clear upper 4K checksum accumulator */
while(b <= ENDUFK ) chksm2=+ *b++;       /* compute upper 4K checksum */
/* b is now pointing to location where first expected
   checksum is to appear
*/
/* get expected checksums from locations in PROM */
excks1= *b++;
excks2= *b;
/* set prom error flags     */
cks1er=0;
cks2er=0;
/* compare actual with expected checksums */
if(chksm1 != excks1) cks1er=1;
if(chksm2 != excks2) cks2er=1;
/* display actual and expected checksums    */
/* convert to ascii (octal) *
atooct(chksm1,ck1);
atooct(chksm2,ck2);
atooct(excks1,exck1);
atooct(excks2,exck2);
/* put strings in output queue */
putttyl(ck1loc);
putttyl(ck1);
putttyl(ex1loc);
putttyl(exck1);
putttyl(ck2loc);
putttyl(ck2);
putttyl(ex2loc);
putttyl(exck2);
/* display no error message if appropriate */
if( (cks1er == 0) && (cks2er == 0) ) putttyl(promok);
/* display error messages if error has been detected */
```

```
else {
   if(cks1er != 0) putttyl(ckerin1);
   if(cks2er != 0) putttyl(ckerin2);
}
putttyl(hitstr);
putttyl(home);
send(); /* empty output queues */
/* wait for break    */
while(*stinstat==0);
c= *d;
initial();       /* go to initialization routine */
}



/* PROMINS - routine for displaying PROM test instructions.
   A brief description of the PROM checksum test is displayed
   on the screen
*/
prmins() {
putttyl(ddl1);
putttyl(prom1);
putttyl(prom2);
putttyl(prom3);
putttyl(prom4);
putttyl(hittst);
box();  /* draw box for target messages */
alarm();        /* fill box with appropriate message */
send(); /* empty output queues */
}



/* RAMTST - routine to test RAM integrity.
   Each word of RAM is tested for its read/write capabilities
   by writing a unique bit pattern in and attempting to read
   the word back. If an error is detected, the routine stops
   and displays the location of the error in octal. The user
   may then continue the test or he may terminate it prematurely
   with appropriate keyboard entries. Upon leaving the test routine,
   the accumulated error count is displayed on the screen in
   decimal. Alternately, a no error message is displayed if there
   were no R/W errors detected during the test.
*/
ramtst() {
int *b,i,k,erflg,ercnt,erloc;
register int *r1,r2,r3; /* define 11-03 registers */
char c,*d;
d=TTIBUF;
ramins();   /* display RAM test instructions    */
/* wait for continue(return) or skip test (any other key)   */
while(*stinstat==0) send();      /* refresh target message while waiting */
c= *d & 0177;
if(c != 015) {   /* check if not CR */
   addfeat();    /* put up additional features list */
   return;
}
/* perform actual Ram test   */
/* disable interupt    */
b=CSR;
```

```
*b=& 0177677
/* put up initial display headings   */
puttty1(dd11);
puttty1(ram1);
send(); /* empty output queues */
erflg=ercnt=0;  /* set error flag and error count to 0 */
r1= STRTRAM;    /* load starting address of RAM in register   */
while(r1 <= ENDRAM) {   /* go till end of RAM */
   r3=052525;   /* load in 0101010101010101 bit pattern */
   r2= *r1;   /* put location in buffer   */
   *r1= r3;      /* put bit pattern into location */
   r3= *r1;      /* read location back into register */
   *r1= r2;   /* restore location   */
   if(r3 != 052525) goto error; /* check for error */
   r2= *r1;   /* put location in buffer */
   r3=0125252;  /* set up 1010101010101010 pattern */
   *r1= r3;      /* load pattern into location */
   r3= *r1;      /* read location back into register */
   *r1= r2;   /* restore location */
   if(r3 == 0125252) goto next; /* check if no error */
/* display error information (location) */
error:   erflg=1;
   ercnt++      /* increment error count */
   erloc=r1;    /* read error location back from register */
/* convert error location to ascii (octal) */
atooct(erloc,errlok);
/* put error location string on output queue */
   puttty1(ramer);
   puttty1(errlok);
   puttty1(rtceh);
   send();       /* empty output queues so error message is displayed */
/* wait for continue(return) or test terminate...  */
   while(*stinstat>=0);
   c= *d & 0177;
   if(c == 015) goto next;       /* check if CR */
/* convert and display the error count (decimal)   */
atodec(ercnt,errcnt);
/* put error count on output queue */
   puttty1(ramcnt);
   puttty1(errcnt);
   send();       /* empty output queues so error count is displayed */
   for(k=0;k<5;k++) for(i=0;i<30000;i++); /* delay so errcnt may be seen*/
   initial();   /* go to initialization routine */
   return;
next:   r1++;    /* go to next word in RAM */
}
/* display end of test messages   */
if(ercnt == 0) puttty1(noram);   /* no error message   */
/* convert and display total error count (decimal) */
else {
   atodec(ercnt,errcnt);
   puttty1(ramcnt);
   puttty1(errcnt);
}
puttty1(hitstr);
send(); /* empty output queues so messages are sent to terminal */
/* wait for any symbol to break */
while(*stinstat>=0)
c= *d;
initial();      /* go to initialization routine */
```

```
}

/* RAMINS - routine to display RAM test instructions
   A brief description of the RAM test is displayed on
   the screen
*/
ramins() {
puttyl(ddl1),
puttyl(ramin1),
puttyl(ramin2),
puttyl(ramin3),
puttyl(ramin4),
puttyl(ramin5),
puttyl(ramin6),
puttyl(hitrt),
box(),  /* draw box for target messages */
alarm()         /* fill box with appropriate message */
send(); /* empty output queues */
}


/* ATODEC - routine to convert word to ascii representation of decimal value.
   This routine takes a 16 bit integer binary word and converts it
   to an ascii string which contains the decimal representation of
   this word. Leading zeros are blanked and the string is left
   justified
*/
atodec(num,asc)
int num;
char asc[7]; {
int i,divide;
for(i=0;i<6;i++) asc[i]=040;    /* fill string with blanks */
asc[6]=0;       /* add eol char at end of string */
/* check for negative number */
if(num<0) {
   num= -num;   /* make positive */
   asc[0]= 45;  /* add negative sign to string */
}
divide=10000;
i=1;
/* check for largest power of 10 which is less than num */
while(divide>num) divide=/ 10;
while(divide>0) {       /* go till ones place is converted */
   asc[i]=num/divide;   /* pull off current decimal place value */
   num=- asc[i]*divide; /* remove this part from num */
   asc[i]=+ 060;        /* convert number to ascii */
   i++; /* increment decimal place counter */
   divide=/ 10; /* go to next power of ten */
}
if(i==1) asc[1]=060;    /* make sure at least on digit is displayed when num=0 */
}


/* ATOOCT - routine to convert word to ascii representation of octal value.
   This routine takes a 16 bit integer binary word and converts it to
   a string of ascii characters which represent its octal value.
   Leading zeros are not blanked; i.e.,6 digits are displayed at all
   times  Numbers are considered to be two's compliment, however,
   the routine is never called with negative numbers in this
   program.
*/
atoocl(num,asc)
int num;
char asc[7]; {
int i;
asc[6]=0;       /* add eol character */
for(i=5;i>0;i--) {      /* convert 5 digits to octal */
   asc[i]=(num & 07) + 060;      /* mask off bits and convert to ascii */
   num=num>>3;  /* shift in next octal digit to 3 least significant bits */
}
asc[i]=(num & 01) + 060;        /* mask off and convert last bit (sign) */
}


/* ALARM - routine to update 'status box' on screen
   This routine displays one of three messages:
   (1) Alarm Disabled
   (2) Target Detected
   (3) No Target Detected
   The routine is only called if it is determined that the box
   needs changing or when the entire screen is changed to a new
   frame in which case the old message is repeated.
*/
alarm() {
switch(almflg) {
case 0: puttty2(aldsl); /* put alarm disabled message in output queue */
        break;
case 1: puttty2(targl); /* put target detected message in output queue */
        break;
case 2: puttty2(notarl);        /* put no target detected message in output queue */
        break;
}
}


/* IROUT - routine to determine if target message needs updating
   This routine is called at the end of the computations and
   determines if the message in the status box is different from
   the one at the end of the previous packet. This saves time by
   not having to change the screen every packet. The audible alarm
   is sounded from this routine every time a target is detected
   unless the audalrm flag is set (from selection of feature 1)
*/
irout() {
int i;
if(aldisab>0) { almflg=0;       /* check if alarm is disabled */
   aldisab--;   /* decrement alarm disable counter */
}
else if(tar != 0) almflg=1;     /* set flag to indicate target detected */
else almflg=2;  /* set flag to indicate no target detected */
/* send audible alarm if proper   */
if(almflg==1 && audalrm==0) puttty2(bell);
/* check if screen should be updated.  */
if(almflg!=oldalfg) {
   oldalfg=almflg;      /* store new screen status for future use in updating */
/* update screen... */
   alarm();      /* this routine updates the status box */
}
}



/* HELP - routine to indicate calculations are lagging behind
   This routine sends a message to the screen if the system is
   being run faster than the computations can keep up with. The
   purpose is mostly to examine just how fast one can push the
   computations. Entered from place in computational routine
   that checks if new set of calculations are being started
   before the current set is done. Routine should get lost
   shortly after this message is displayed due to use of stack
   in computations
*/
help() {
puttty2(help1);
send(); /* empty output queues */
}



/* BOX - routine to draw status box in workspace.
   This routine constructs the box at the bottom of the
   workspace area that is used to highlight important info
   about the current status of the detector such as if
   the alarm is disabled or if a target has or hasn't
   been detected
*/
box() {
puttty1(box1);
puttty1(star21);
puttty1(box2);
puttty1(star2);
puttty1(box3);
puttty1(star2);
puttty1(box4);
puttty1(star2);
puttty1(box5);
puttty1(star21);
}



/* DMATST - routine to test operation of DMA board.
   This routine is used to enter an assembly language
   routine which actually does the testing of the DMA.
   There are seven separate tests which are performed
   in the assembly language routine
*/
dmatst() {
char c,*d;
d=TTIBUF;
loop: dmats();    /* go to routine which sets up for call to assembly code*/
        while(*stinstat)=0);     /* wait for repeat character */
        c= *d;
      goto loop; /* go back and perform DMA test again */
}
```

```
/* DMATS - routine to prepare for and call assembly language DMA test routine.
   This routine initializes a few flags which are used to interface
   between the assembly language code and here and also provides
   for the display of test results on the screen. The assembly language
   routine tests seven functions of DMA operation.
   (1) R/W of WCR
   (2) R/W of BAR
   (3) R/W of DBR
   (4) Function-status lines
   (5) Response to bus INIT signal
   (6) Generation of interrupt request
   (7) Transfering of data into memory
   It is assumed that the user has inserted the DMA maintenance
   cable between the two DMA ports. This provides the feedback between
   several lines that is neccessary for proper execution of several
   of the tests.
*/
dmats() {
int *b;
/* perform actual DMA test    */
/* disable dma interupts    */
b=CSR;
*b=& 0177677;
/* label screen */
puttty1(ddl1);
puttty1(DMA11);
puttty1(hitrep);
send(); /* empty output queues */
/* set error flags to 0. These will be set to a 1 during the
   assembly language test routine if an error is detected */
erWCR=erBAR=erDBR=erRST=erFCT=0;
dma();    /* go to assembly language DMA test routine ... */
/* display results   */
if(erWCR>0) puttty1(WCRer);      /* error in R/W of WCR */
else puttty1(noWCR);    /* no error in R/W of WCR */
if(erBAR>0) puttty1(BARer);      /* error in R/W of BAR */
else puttty1(noBAR);    /* no error in R/W of BAR */
if(erDBR>0) puttty1(DBRer);      /* error in R/W of DBR */
else puttty1(noDBR);    /* no error in R/W of DBR */
if(erRST>0) puttty1(RSTer);      /* improper response to INIT */
else puttty1(noRST);    /* proper response to INIT */
if(erFCT>0) puttty1(FCTer);      /* error in function-status lines */
else puttty1(noFCT);    /* no error in function-status lines */
if(erINT>0) puttty1(INTer);      /* no interrupt detected */
else    puttty1(noINT); /* no interrupt detected */
if(erTRN>0) puttty1(TRNer);      /* error in data transfer */
else    puttty1(noTRN); /* no error in data transfer */
send(); /* empty output queues so error messages appear on screen */
}



/* DMAINS - routine to display DMA test instructions
   This routine displays a brief description of the DMA test
   routine and some instructions on how to perform the test.
   The execution of the DMA test requires the insertion of the
   DMA maintenance cable.
*/
dmains() {
puttty1(ddl1);
puttty1(dma1);
puttty1(dma2);
puttty1(dma3);
puttty1(dma4);
puttty1(dma5);
puttty1(dma6);
puttty1(dma7);
puttty1(dma8);
puttty1(hithlt);
box();  /* draw box for status messages */
alarm();        /* fill box with current status message */
puttty1(home);  /* reposition cursor */
send(); /* empty output queues */
}


/* BSS - routine to place 'Type L to see list of...' line on screen.
   This routine clears the bottom of the screen (monitor) and sends
   the 'Type L to see list of additional features.' line. The main
   purpose of this routine is to save prom space by avoiding
   repeated subroutine calls when this function is needed.
   Three for the price of one....
*/
bss() {
puttty1(blnk);
puttty1(seelst);
puttty1(sk21);
}


/* BDS - routine to place 'Invalid parameter or command.' line on screen
   This routine clears the bottom of the screen (monitor) and sends
   the invalid parameter value line to the screen. The reasoning
   behind this routine is identical to the BSS routine - namely to
   save space in PROM.
*/
bds() {
puttty1(blnk);
puttty1(daterr);
puttty1(sk21);
}
```

```
/ ***********************************************************
/ **                                                       **
/ ** ASSEMBLY LANGUAGE ROUTINES -                           **
/ ** This section of code contains all assembly language routines **
/ ** used in the IRCCD intrusion detector alarm program. These    **
/ ** include                                               **
/ ** (1) 'C' runtime startoff routines                     **
/ ** (2) Other routines that 'C' calls                     **
/ ** (3) Interupt service routines                         **
/ ** (4) Computational routine to actually implement algorithm **
/ ** (5) DMA test routine                                  **
/ ** (6) Routine for computing threshold table             **
/ ** (7) Routine to execute 'HALT' instruction from 'C'    **
/ **                                                       **
/ ***********************************************************


/ Define instructions that are normally not in the assemblers vocabulary

        rti = 000002
        nop = 000240
        reset = 000005
        mtps = 106404
        halt = 000000


/ C runtime startoff routines

 text
 globl  _exit
 globl  _main
 globl  _end
 globl  _edata
 globl  _start

/ Set start of program to 20000 octal

         =   + 20000
_start  mov     $400,r0         /setup vector area with halts
1       clr     -(r0)
        mov     r0,-(r0)
        tst     r0
        bne     1b

        mov     $_edata,r0      /clear bss area (uninitialized data)
2       cmp     r0,$_end
        beq     3f
        clrb    (r0)+
        br      2b
3

/ Set stack pointer to high RAM

        mov     $0160776,sp

        mov     sp,r0
        mov     (r0),-(sp)
        tst     (r0)+
        mov     r0,2(sp)
        jsr     pc,*main
        cmp     (sp)+,(sp)+
        mov     r0,(sp)
        jsr     pc,*$_exit
        sys     exit


/ 'C' register save and restore -- version 12/74
/ Called at entry to and exit from 'C' routines

.globl  csv
.globl  cret

csv:
        mov     r5,r0
        mov     sp,r5
        mov     r4,-(sp)
        mov     r3,-(sp)
        mov     r2,-(sp)
        tst     -(sp)
        jmp     (r0)

cret:
        mov     r5,r1
        mov     -(r1),r4
        mov     -(r1),r3
        mov     -(r1),r2
        mov     r5,sp
        mov     (sp)+,r5
        rts     pc


/ Interrupt handling routines

.globl _endpk
.globl _dumm


/ ENDPK - routine to service the interrupt occuring at end of data packet
/ This routine is entered [illegible] an interrupt from the DMA
/ heralding that a data transfer has been completed. The DMA is then
/ reset in anticipation of the next packet and the assembly language
/ computational routine is entered. This routine also determines
/ where the next packet of data is to be loaded, if the computations
/ are getting behind, and calls the 'C' routine that displays the
/ proper status message

_endpk  mov     r0,-(sp)        / save register 0
        mov     r1,-(sp)        / save register 1

/ Check if falling behind

        tst     _flag
        beq     setflg

/ Print error message

        jsr     pc,_help
setflg  mov     $1,_flag
```

```
/ Toggle function 3 line to stop start convert pulses

        bic     $10,*$172414
        nop                     / wait a while before switching f3 line back
        bis     $10,*$172414
        mov     $177400,*$0172410           /load wcr register with -256

/ Set go bit for next load

        bis     $101,*$172414

/ Determine which buffer to load next packet into

        tst     *bufnum
        bne     rbf2

/ Load next one into rbuf2

        mov     $1,*bufnum
        mov     $*rbuf2,*$0172412           /load BAR
        mov     $*rbuf1,*rstr   / set pointer to current buffer
        br      cmpt

/ Load next packet into rbuf1

rbf2    clr     *bufnum
        mov     $*rbuf1,*$0172412           / load BAR
        mov     $*rbuf2,*rstr   / set pointer to current buffer

/ Check if computations should be skipped due to parameter change

cmpt:   tst     *intfle
        beq     crn
        inc     *aldisab            / disable alarm for at least one packet
        br      nocrn

/ Call computational routine  crunch

crn:    jsr     pc,*crunch

/ Call  C  routine which updates status box on screen

nocrn:  jsr     pc,*irout
        clr     *flag   / clear flag used in determining if calculations slow

/ Restore r1 and r0

        mov     (sp)+,r1
        mov     (sp)+,r0
        rti


/ DUMMY - a "almost nothing" interrupt service routine
/ This is a routine that is entered only at system initialization
/ when the DMA interrupts are first enabled. At that time, an interrupt
/ is generated by the DMA (since it is in "ready" state) but this
/ interrupt does not signal the end of a data packet. Hence, this
/ routine merly sets the interrupt vector pointing to the ENDP
/ routine and thus prevents entering the computations before an
```

```
 actual data packet is received

+dummy  mov     $+endpk *$0124

 Set initial values of function lines

        bic     $6,*$172414
        bis     +tref,*$172414
        rti



 IRCCD Intrusion Detector Alarm   computational routines
/ CRUNCH - IRCCD intrusion detector alarm computational routines
/ This routine is the heart of the detector  It performs all
/ computations and decisions dictated by the original algorithm
/ The routine is divided into three main sections
/ (1) Computation of background spatial average and initiating
/     stare time changes.
/ (2) Computation of background time average.
/ (3) Threshold comparison and target - no target decision
/ Through out this routine there are many strange looking
/ quantities which were initialized elsewhere in some 'C
/ routines  The purpose of these intermediate variables is
/ to save time in this critical section of code  These will
/ be explained as they are used
/ There are several main vectors that are refered to in the
/ computational routine  These are
/ (1) 'x' - this vector contains the past N packets of data
/     samples  1 Pth of these samples are updated each iteration
/     through the calculations  The vector is set up as a circular
/     queue of length 256*N where the last element is thought
/     of as being immediately followed by the first  This enables
/     one pointer to access both the locations of the newest entry
/     and the oldest entry  'xptr' is this pointer and is left
/     pointing to the location of the oldest data sample after each
/     pass through the routine
/ (2) 'm' - this vector contains several different quantities  The
/     first 256 elements contain the low order word of the accumulated
/     double prescision sum of past (and current) data samples  The
/     second 256 words contain the background time average (a divided
/     version of the first 256 elements of 'm')  The third 256
/     words contain the decision threshold corresponding to the
      background time average stored in the second 256 words  The
/     last 256 words contain the high order word of the accumulated
/     double prescision sum of past (and present) data samples
/     This strange structure is present to allow for faster
/     computations.
/ (3) 'a' - this is a vector of 256 words that contains the past
/     history of threshold violations  Each word corresponds to
/     a cell of the IRCCD array while various bits correspond  to
/     the past history of threshold violations  The second bit of
/     each word represents the threshold violation information
/     of the current packet, the third bit corresponds to one packet
/     into the past, the fourth bit corresponds to two packets into
/     the past, etc  This format greatly simplifies the target-no target
/     decision process
```

```
 .globl _crunch
 .text
_crunch:
        jsr     r5,csv  / call register save routine
        mov     r5,_tmp2        / save r5

/Compute background spatial average

/ 'ntbkspav' is a number that equals the number of terms in the
/ background spatial average, i.e., 256 divided by D

        mov     _ntbkspav,r0

/ Set r1 pointing to start of incoming data buffer
/ 'yetr' was set in ENDPK interrupt service routine

        mov     _yetr,r1

/ 'tbad' is equal to 2*D. Since D is used to increment a word
/ address, it must be multiplied by two to account for two
/ bytes to a word

        mov     _tbad,r4
        clr     r3
        clr     r2
loopa:  add     (r1),r3 / add sample into accumulator
        adc     r2      / add carry to compute double prescision sum
        add     r4,r1   / set r1 to point to next Dth word
        sob     r0,loopa        / go back till all terms are in accumulator

/ Divide sum by D to compute background spatial average
/ Since all possible D values are powers of 2, division is done
/ by shifting the accumulated double prescision sum. 'dshft'
/ tells how many places to shift for the current D value

        ashc    _dshft,r2

/ Round off result after divide (shift)

        adc     r3

/ Background spatial average is now in r3
/ Check if should test for stare time change
/ Stare time change should only be checked if a change was
/ not performed the previous time. 'fstat' keeps track
/ of this status

        tst     _fstat
        beq     ckst    / go to check for stare time change
        clr     _fstat
        jmp     date    / do not check for stare time change

/ Check for stare time change
/ Set r0 pointing to start of C2 value table

ckst:   mov     $_cc2,r0

/ Choose element in table corresponding to stare time that was in
/ use when current data packet was formed. 'oldtref' contains this
/ information
```

```
        add     *oldtref,r0

/ Check for stare time decrease

        cmp     r3,(r0)
        blt     stinc

/ Decrease stare time if possible

        inc     *fstat  / set flag indicating stare time change

/ Test if stare time can be decreased any further

        tst     *tref
        beq     date    / if not, do not change
        sub     $2,*tref        / change tref (not sent out yet)

/ Go to code which sends out new stare time via function lines 1 2

        jmp     update

/ Check for stare time increase

stinc   mov     $cc1,r0         / set r0 pointing to start of C1 threshold table

/ Choose element which corresponds to stare time used when current
/ packet was formed

        add     *oldtref,r0
        cmp     r3,(r0) / check for increase
        bgt     date

/ Increase stare time if possible

        inc     *fstat  / set flag to indicate stare time change
        cmp     $6,*tref        / check if 'tref' at maximum value
        beq     date    / do not change if at maximum
        add     $2,*tref        / increase stare time

/ Send updated stare time to outside world
/ Set function 1 and 2 lines to newly computed value
/ These lines are accessed via two bits in the CSR of the DMA

update  bic     $6,*$172414
        bis     *tref,*$172414

/ Update past history of stare times  Must keep track
/ two packets into the past

date    mov     *trefm1,*oldtref
        mov     *tref,*trefm1


/ Computation of background time average
/ The computations for the background time average are divided
/ up so that roughly 1 Pth of the computations are done each
/ time  There are three parameters computed in  C  which determine
/ exactly how these computations are divided up  These are
/ 'pnorm','plast', and 'nl'  There are  pnorm  calculations
/ performed 'n1'-1 iterations and 'plast' calculations performed
/ the 'n1' iteration. There are no calculations performed the
/ last P-'n1' iterations.

/ Increment counter to keep track of which Pth of calculations on

btavs   inc     *pcnt

/ Check to see if wrap around on circular queue storing past sample value

        mov     *xptr,r0
        sub     $*x,r0
        cmp     r0,*wrap
        blt     nowrap
        mov     $*x,*xptr
nowrap  mov     *yptr,r0        / set r0 pointing to start of current packet
        add     *yincr,r0       / increment to proper Pth of packet
        mov     *xptr,r1        / set r1 pointing to proper Pth of past samples
        mov     *mptr,r2        / set r2 pointing to proper Pth of accum sum

/Determine if 'n1' P-th of background time average

        cmp     *pcnt,*n1

/ Go to section of code that performs 'plast' calculations

        beq     last

/ Go to section of code that performs no calculations

        bgt     chklst

/ Go to section of code that performs 'pnorm' calculations

        jmp     norm

/ Perform last section of calculations

last    mov     *plast,r3       / set counter to 'plast'

/ Subtract oldest past sample from accummulated double prescision sum

loopc   sub     (r1),(r2)
        sbc     3000(r2)        / make double prescision subtraction

/ Replace past sample in 'x' with current sample

        mov     (r0),(r1)+

/ Add current sample into accumulated double prescision sum

        add     (r0)+,(r2)
        adc     3000(r2)        / make double prescision sum

/ Move double prescision sum into r4,r5 in preparation for divide

        mov     3000(r2),r4
        mov     (r2)+,r5

/ Divide sum by N to get background time average
```

```
        div     *bign,r4

/ Round off after divide (rdthr is set in C routine

        cmp     r5,*rdthr
        blt     nord
        inc     r4        / round off

/ Store background time average in second 256 words of  m

nord    mov     r4,776(r2)

/ The background time average will now be used to look up the
/ appropriate threshold in a table ('btab') which is pointed to
/ by  bstr  This table contains thresholds corresponding to
/ background time averages of between 0 and 1023  This is alright
/ for the two longest stare times but when operating at the two
/ shortest stare times the values may go as high as 4095 Thus,when
/ using the two shortest stare times, the thresholds are looked
/ up by first dividing the background time average by four, looking
/ up in the table, and then multiplying this result by two  This
/ proceedure works since the thresholds are proportional to the
/ square root of the background time average

/ Check whether using one of two longest or two shortest stare times

        cmp     *oldtref,$3
        bgt     noshfl  / using one of two longest stare times

/ Using one of two shortest stare times  At this point the result
/ is only divided by two (instead of 4) since it will be used to
/ look up an element in a word table (must increment two bytes for
/ each word)  This is equivalent to looking up the value corresponding
/ to the background time average divided by four

        asr     r4
        bic     $1,r4   / make sure even address
        add     *bstr,r4        / set r4 pointing to correct element in table
        mov     (r4),r5 / move threshold into r5
        asl     r5      / multiply threshold by 2 (since divided by 4 before
        mov     r5,1776(r2)     / move threshold into third 256 words of  m
        br      retrnl  / go back for next iteration

/ Look up threshold for longest two stare times

noshfl  asl     r4      / mult by two to make word address
        add     *bstr,r4        / set r4 pointing to proper element in table
        mov     (r4),1776(r2)   / move threshold into third 256 elements of  m
retrnl  sob     r3,loopc        / go back for next iteration

/ Reset pointers

        mov     r1,*xptr
        mov     $*m,*mptr
        clr     *yincr

/ Check if Pth iteration

chklst  cmp     *pcnt,*bigp
        bne     decd    / go to decision routine
```

```
/ Reset rcnt

        clr     *rcnt
        br      decd

/ Perform 'norm' segment of calculations
/ These calculations are essentially identical to those in the
/ 'plast' section of code. The code is repeated to increase speed
/ For comments, see 'plast' section

norm    mov     *enorm,r3
loopb   sub     (r1),(r2)
        sbc     3000(r2)
        mov     (r0),(r1)+
        add     (r0)+,(r2)
        adc     3000(r2)
        mov     3000(r2),r4
        mov     (r2)+,r5
        div     *bign,r4
        cmp     r5,*rdlnsh
        blt     nornd
        inc     r4
nornd   mov     r4,776(r2)
        cmp     *oldtref,$3
        bgt     noshf2
        asr     r4
        bic     $1,r4
        add     *bstr,r4
        mov     (r4),r5
        asl     r5
        mov     r5,1776(r2)
        br      retrn2
noshf2  asl     r4
        add     *bstr,r4
        mov     (r4),1776(r2)
retrn2  sob     r3,loopb

/ Reset pointers

        mov     r1,*xptr
        mov     r2,*mptr
        add     *tenorm,*xtncr



/ Decision making section of computations
/ See comments at start of computational routine for description
/ of 'a' vector

decd    mov     $*m,r1
        add     $1000,r1          / r1 points to background time average
        mov     *sptr,r2          / r2 points to current sample value
        mov     $400,r3 / r3 is counter to keep track of spatial location
        mov     $*a,r4  / r4 points to 'a' vector
        clr     r5
        mov     sp,*tmp1          / save stack pointer
        clr     *tar    / clear target flag
loopd   mov     (r1)+,r0          / put background time average into r0
        sub     (r2)+,r0          / subtract current sample value
```

```
/ Take the absolute value of difference

        bpl     pos
        neg     r0

/ Shift bits in (r4) to make room for current threshold violation
/ information

pos:    asl     (r4)

/ Compare threshold with absolute value of diffence of current sample
/ and background time average

        cmp     77(r1),r0
        bgt     nobis   / test if no threshold violation

/ Set second bit to indicate threshold violation at current time instant

        bis     $2,(r4)

/ Mask off n divided by 3 most recent threshold decisions
/ 'mask' is set in 'C' routine

nobis   bic     *mask,(r4)

/ Move history into r0

        mov     (r4)+,r0

/ Make r0 point to location in table 'tab' that contains a number
/ which equals the total number of threshold violations in the
/ past n divided by 3 most recent packets

        add     $*tab,r0
        mov     (r0),-(sp)      / move this number on to stack

/ r5 contains the sum of the total number of threshold violations
/ in the past n divided by 3 packets of the two spatially
/ previous cells of the IRCCD array

        add     (sp),r5 / add violations in present spatial cell

/ Do not make target decision for first two cells

        cmp     $376,r3
        blt     init

/ Compare t to the sum of the number of threshold violations in the
/ three spatially adjacent cells which have occured during the last
/ n divided by 3 packets

        cmp     *smt,r5
        bgt     targa   / skip setting target detected flag
        mov     $1,*tar / set target detected flag

/ Subtract number of threshold violations in the IRCCD cell
/ two positions away from the current cell

targa   sub     4(sp),r5
init    sob     r3,loopd        / go to next spatially adjacent cell in array
```

```
        jmp     here    / space filler
here    mov     _tmp1,sp        / restore stack pointer
        mov     _tmp2,r5        / restore r5
        jmp     cret    / go to register restore routine


/ THRESH - routine for setting up threshold table
/ This routine computes the threshold table that is used in the
/ target calculations  The algorithm for computing square roots
/ is a Newtons method type scheme using a tedious form of
/ fixed point arithmetic  The factor dependent on N was found by
/ table lookup in a 'C' routine (sgn)  The d parameter is assumed
/ to be in a nnnnn nnnnnnn binary format (7 binary places)
/ The iterative portion is used to compute sqrt(a) where a corresponds
/ to the background time average

 globl  _thresh
 text
_thresh jsr     r5,csv
        mov     r5,_tmp3
        mov     $2000,r5        /no  of elements in table
        clr     r4
loop1   mov     $77776,r3       /r3 contains initial guess for x
        ash     $4,r4           /r4 contains shifted value of a
        mov     $10,r2          /set no  of iterations for N.R.
loop2   mov     r3,r0
        mul     r3,r0           /r0,r1 contains x**2
        sub     r4,r0           /r0,r1 contains f(x)
        div     r3,r0           /r0 contains f(x) over f'(x)
        inc     r0              /round
        asr     r0              /set b.p. to match x
        sub     r0,r3           /r3 contains new x value
        sob     r2,loop2
        ash     $-3,r4          /r4 contains 2*a

/ Now must multiply three terms together to get threshold

        mov     r3,r0
        mul     _sgn,r0
        ashc    $1,r0
        mul     _smd,r0
        ashc    $1,r0
        inc     r0
        asr     r0              /r0 now contains desired thresh
        mov     _bstr,r1
        add     r4,r1           /set address in table
        asr     r4              /return a to normal
        inc     r4
        mov     r0,(r1)         /put entry in table
        sob     r5,loop1
        clr     _btab
        mov     _tmp3,r5
        jmp     cret


/ DMA - routine to actually perform the test on the DMA board
/ This routine is entered from the 'C' routine DMATS  Test results
/ are returned to 'C' by setting error flags  There are seven DMA
/ functions which are tested
/ (1) R-W of WCR
/ (2) R-W of BAR
/ (3) R-W of DBR
/ (4) Response to bus INIT signal
/ (5) Function-Status lines
/ (6) End of transfer interrupt generation
/ (7) Data transfer capabilities

 globl _dma
 text
_dma    jsr     r5,csv
        mov     $200,r4
        mtps                    /disable system interupts

/ Test r-w of various registers

        mov     $172410,r0      /load WCR address
        mov     $52525,r1       /load test pattern
        mov     r1,(r0) /write test pattern into WCR
        mov     (r0),r2 /read WCR back
        cmp     r1,r2   /check if error
        beq     WCRok1
        inc     _erWCR  /set error flag
WCRok1  com     r1      /try new pattern
        mov     r1,(r0) /write into WCR
        mov     (r0)+,r2        /read WCR back
        cmp     r1,r2   /test if error
        beq     WCRok2
        inc     _erWCR  /set error flag
WCRok2  mov     r1,(r0)         /test BAR
        mov     (r0),r2 /read back from BAR
        bic     $1,r2   /ignore first bit (always reads as a 1)
        cmp     r1,r2   /test if error
        beq     BARok1
        inc     _erBAR  /set error flag
BARok1  com     r1      /try new bit pattern
        mov     r1,(r0) /write into BAR
        mov     (r0)+,r2        /read from BAR
        cmp     r1,r2   /test if error
        beq     BARok2
        inc     _erBAR  /set error flag
BARok2  bis     $1,r1           /test DBR
        inc     r0      /make r0 point to DBR
        inc     r0
        mov     r1,(r0) /write bit pattern into DBR
        mov     (r0),r2 /read bit pattern from DBR
        cmp     r1,r2   /test if error
        beq     DBRok1
        inc     _erDBR  /set error flag
DBRok1  com     r1      /try new bit pattern
        mov     r1,(r0) /write bit pattern into DBR
        mov     (r0),r2 /read bit pattern back from DBR
        cmp     r1,r2   /test if error
        beq     DBRok2
        inc     _erDBR  /set error flag
DBRok2  mov     $172410,r0      /test if reset clears appropriate bits
        clr     r1

/ Set all possible bits in various registers
```

```
        mov     $-1,(r0)+
        mov     $-2,(r0)+
        mov     $-3,(r0)+
        mov     $-1,(r0)
        mov     $172410,r0      /set r0 pointing to WCR
        reset   /issue reset command (bus INIT)

/ Test if proper bits are cleared in various registers

        mov     (r0)+,*expec
        bne     RSTer
        mov     (r0)+,*expec
        bic     $1,*expec
        bne     RSTer
        mov     (r0)+,*expec
        cmp     $200,*expec
        bne     RSTer
        mov     (r0),*expec
        beq     RSTok
RSTer   inc     *erRST  /set error flag

/ Test function - status lines by sending out all possible
/ combinations of function line values and determining if
/ proper values are read back by status lines.

RSTok   mov     $7000,*expec    /set up expected status line values
        mov     $16,r0  /set initial values of function lines
        mov     $172414,r2      /set r2 to point to CSR
FCT     mov     r0,(r2) /send out function lines
        mov     (r2),*tmp1      /read CSR
        bic     $170777,*tmp1   /mask off status bits
        cmp     *tmp1,*expec    /test if error
        beq     FCTok
        inc     *erFCT  /set error flag
FCTok   sub     $1000,*expec
        sub     $2,r0
        bpl     FCT     /go to next set of f line values

/ Test DMA data transfers and end of load interupts

        mov     $-1000,*$172410         /set up WCR
        mov     $*ybuf1,*$172412        /set up BAR

/ Put something interesting in ybuf1

        mov     $*ybuf1,r0
        mov     $400,r1
load    mov     r1,(r0)+
        clr     (r0)+
        sob     r1,load
        mov     $*dmaint,*$124          /load int  service address
        clr     *flag

/ Enable system interupts

        clr     r4
        mtps

/ Hit maint [illegible] and ie bits
```

```
        mov     $100101 *$172414
        bis     $400,*$172414           / hit cycle request

/ Go into wait loop

        clr     *exrec
wate    inc     *exrec
        bne     wate

/ Test how things went .
/ Check for interupt

        tst     *flag
        bne     INTok
        inc     *erINT  /set error flag
INTok   mov     $*xbuf1,r0
        mov     $400,r1
TRN     mov     (r0)+,r2
        cmp     r2,(r0)+         test if proper word transfered into memory
        bne     TRNer
        sob     r1,TRN
        br      TRNok
TRNer   inc     *erTRN  /set data transfer error flag
TRNok   reset
        jmp     cret


/ DMAINT - interrupt service routine used to set flag indicating
/ interrupt received during testing of DMA board

 globl  *dmaint
 text
*dmaint mov     $1,*flag          set flag
        rti


/ HALT - routine to be called from 'C' to execute a 'halt'
/ instruction. This is used to stop the proccessor when it is
/ desired to enter the DMA test routine. The halt allows one
/ to insert the DMA maintenance cable

 globl  *halt
 text
*halt   halt
```

```
/ Checksum Test Routine

        mov     $20,r0          /clear checksum areas in RAM
        mov     $65000,r1
loopa   clr     (r1)+
        sob     r0,loopa
        mov     $20000,r3       /load start of PROM address
        mov     $65000,r2       /load start of checksum address
        mov     $10,r1          /load number of K words counter
loopb   mov     $2000,r0        /load number of bytes counter
loopc   movb    (r3),r4         /get low order byte
        add     r4,(r2)         /compute low order checksum
        inc     r3              /go to high order byte
        movb    (r3),r4         /get high order byte
        add     r4,2(r2)        /compute high order checksum
        inc     r3              /go to next word
        sob     r0,loopc
        add     $4,r2           /move pointer to store next 1K checksums
        halt
```

```
021550t BARok1     024662t L20018     041642t L20127     025270t L283
021566t BARok2     025006t L20020     042360t L20128     025314t L286
021612t DBRok1     025134t L20022     042404t L20129     025374t L287
021630t DBRok2     025230t L20024     042430t L20130     025512t L292
021752t FCT        025326t L20026     042454t L20131     025500t L293
022002t FCTok      025432t L20028     042500t L20132     057750d L3
022126t INTok      025310t L20029     042524t L20133     026660t L300
057744d L1         026076t L20032     042550t L20134     025640t L301
057770d L10        026152t L20034     023044t L202       025670t L302
022240t L10000     026226t L20036     023054t L203       026604t L303
057646d L10002     026302t L20038     023064t L204       025700t L305
057672d L10004     026356t L20040     023074t L205       026062t L307
023702t L10005     026456t L20042     023104t L206       025752t L308
023672t L10006     025600t L20045     023114t L207       026016t L314
024164t L10007     030562t L20047     023124t L208       026050t L316
024134t L10008     031056t L20049     023134t L209       026512t L318
024332t L10009     031172t L20051     023164t L210       026136t L319
030774t L10010     031232t L20053     023214t L213       026110t L323
031402t L10011     031444t L20055     023314t L217       026212t L325
032112t L10012     031616t L20060     023320t L219       026164t L329
032332t L10013     031666t L20062     023336t L220       026266t L331
032322t L10014     031674t L20064     023426t L222       026240t L335
032636t L10015     031742t L20066     023436t L223       026342t L337
033026t L10016     032146t L20068     023402t L225       026314t L341
033304t L10017     032374t L20070     023534t L231       026416t L343
057720d L10019     032672t L20072     023644t L233       026370t L347
057736d L10021     033062t L20074     023610t L234       026476t L350
057772d L11        033142t L20076     023640t L236       026464t L353
057774d L12        033340t L20078     023710t L238       026644t L357
057776d L13        033420t L20080     023742t L240       026740t L361
022212t L175       033642t L20082     023766t L242       027002t L365
022444t L179       034020t L20083     024004t L243       027510t L370
022450t L180       035020t L20085     025212t L246       030534t L397
022574t L184       035150t L20087     024214t L247       030540t L398
022652t L185       035310t L20089     024210t L249       030602t L399
022676t L188       035446t L20091     024100t L250       057752d L4
022704t L189       035606t L20093     024114t L251       030670t L402
022712t L190       036416t L20095     024172t L252       030676t L403
022720t L191       036524t L20097     024250t L254       031010t L406
022726t L192       036520t L20099     024274t L255       031014t L407
022734t L193       023034t L201       024742t L256       031076t L408
022744  L194       036566t L20101     024516t L259       031210t L411
022754t L195       037276t L20103     024504t L260       031416t L419
022764t L196       037440t L20105     024500t L261       031422t L420
022774t L197       037472t L20107     024626t L263       031464t L421
057746d L2         040046t L20108     024634t L264       031640t L425
023024t L200       040252t L20110     024622t L265       031712t L427
022206t L20001     040640t L20112     024616t L266       032126t L435
022344t L20003     040634t L20114     024702t L269       032132t L436
022374t L20005     041006t L20115     025044t L272       032166t L437
022334t L20006     040400t L20117     025026t L273       032346t L442
023354t L20008     040752t L20118     025070t L276       032352t L443
023472t L20010     041156t L20120     025110t L277       032414t L444
024056t L20012     041262t L20122     025126t L278       032530t L447
024400t L20014     041312t L20124     025156t L279       032652t L450
024532t L20016     041504t L20126     025216t L282       032656t L451
```

```
032712t L452       040320t L595       041612T ←alarm     044462D ←c2lst
033042t L457       040676t L596       062010B ←aldisab   032774T ←c2srv
033046t L458       040444t L598       051664D ←aldsl     143670B ←c2str
033102t L459       040672t L599       144030B ←almfls    062040B ←ccl
033320t L467       057756d L6         143264B ←aptr      062026B ←cc2
033324t L468       040534t L600       024014T ←asclon    143742B ←chksm1
033360t L469       040720t L608       143714B ←atdbin    143744B ←chksm2
033614t L477       040772t L610       143704B ←atdoct    143746B ←ck1
033610t L478       041454t L617       035200T ←atod      050430D ←ck1loc
033572t L480       041244t L621       052700D ←atod1     143756B ←ck2
033620t L481       041300t L622       052746D ←atod2     050440D ←ck2loc
033626t L482       041424t L624       053044D ←atod3     050544D ←ckerin1
033634t L483       041646t L635       143740B ←atodbyt   050630D ←ckerin2
033646t L484       041636t L638       041142T ←atodec    051362D ←clscrn
033704t L486       041652t L639       035466T ←atodins   043302D ←cmdno
033712t L488       041660t L640       034774T ←atodtsl   043336D ←cmds
033720t L489       042010t L643       041460T ←atooct    143600B ←command
033726t L490       041714t L644       143604B ←audalrm   043500D ←crlf
034014t L494       041740t L645       043000T ←bds       020334T ←crunch
057754d L5         041732t L646       043476D ←bell      043442D ←ctrb
035174t L504       041766t L648       143562B ←bfpti1    043460D ←ctrx
035024t L505       042206t L659       143564B ←bfpti2    143576B ←datares
035064t L507       042212t L660       143566B ←bfptot1   046532D ←dater2
035154t L509       042354t L665       143570B ←bfptot2   046442D ←daterr
035262t L514       042400t L667       143606B ←bisd      143620B ←datstrn
036172t L527       042424t L669       143634B ←bisdstr   044634D ←dd11
035612t L528       042450t L671       132070B ←bisn      045426D ←dd110
035652t L530       042474t L673       143640B ←bisnstr   045526D ←dd112
036002t L531       042520t L675       136076B ←bisp      045650D ←dd114
036152t L532       042544t L677       143644B ←bispstr   045770D ←dd116
036136t L533       057760d L7         047656D ←binloc    046110D ←dd118
037062t L539       057763d L8         043504D ←bksp      046230D ←dd119
036422t L540       057766d L9         043372D ←blank     044726D ←dd12
036464t L542       021730t RSTer      046422D ←blnk      044734D ←dd13
036636t L551       021734t RSTok      042040T ←box       045006D ←dd14
036624t L552       022136t TRN        051400D ←box1      045124D ←dd15
036676t L554       022150t TRNer      051420D ←box2      045166D ←dd16
036722t L556       022154t TRNok      051430D ←box3      045306D ←dd18
036766t L557       021512t WCRok1     051440D ←box4      030152T ←ddlist
036750t L558       021530t WCRok2     051450D ←box5      043510D ←ddloc
037004t L561       052034D ←BARer     042740T ←bss       043776D ←ddlst
037050t L562       057642D ←CKSM1     132074B ←bstr      030472T ←ddsrv
037032t L563       057644D ←CKSM2     137122B ←btab      023452T ←decode
040116t L571       052110D ←DBRer     143572B ←btcnt1    046244D ←disp2
037302t L572       051740D ←DMAll     143574B ←btcnt2    046604D ←disp3
037344t L574       056736D ←Err       144034B ←buf       046670D ←disp3b
037454t L575       052340D ←FCTer     060002B ←bufnum    034530T ←displa
037506t L577       052444D ←INTer     143612B ←c1        030270T ←dlist
037566t L579       056706D ←Noer      030440T ←c1list    043612D ←dloc
037604t L580       052164D ←RSTer     043746D ←c1loc     044222D ←dlst
040016t L581       052604D ←TRNer     044550D ←c1lst     021454T ←dma
040052t L582       051760D ←WCRer     033252T ←c1srv     054010D ←dma1
040034t L583       043056T ←←cleanu   143676B ←c1str     054054D ←dma2
040076t L585       136104B ←a         143614B ←c2        054152D ←dma3
041012t L592       033734T ←addfeat   030406T ←c2list    054206D ←dma4
040256t L593       047704D ←adtsl     043716D ←c2loc     054310D ←dma5
```

```
054362D ←dma6      047140D ←line2     050072D ←pdsdis2   057156D ←spcer
054430D ←dma7      047216D ←line3     036176T ←pdsins    137116B ←sgn
054502D ←dma8      047274D ←line4     035562T ←pdslst    051506D ←star2
042564T ←dmains    047322D ←line5     132066B ←plast     051460D ←star21
022162T ←dmaint    047352D ←line6     136100B ←pnorm     020000T ←start
042230T ←dmats     047402D ←line7     030236T ←pplist    143272B ←stinsta
042170T ←dmatst    047446D ←line8     043564D ←pploc     043060D ←string
143270B ←drvcsr    046760D ←line9     044136D ←pplst     043202D ←tab
062022B ←dshft     033530T ←lsrv      031354T ←ppsrv     137106B ←tar
032066T ←dsrv      132076B ←m         050236D ←prmdis1   051534D ←targ1
020310T ←dummy     022174T ←main      050264D ←prmdis2   062020B ←tbgd
060000D ←edata     137110B ←mask      050320D ←prmdis3   037066T ←termins
144046B ←end       050202D ←mkmon     040122T ←prmins    036372T ←termtst
020122T ←endpk     050150D ←mkwor     055226D ←prom1     050140D ←test
143124B ←erBAR     132062B ←mptr      055274D ←prom2     021316T ←thresh
143126B ←erDBR     132064B ←nl        055370D ←prom3     030354T ←tlist
143134B ←erFCT     030322T ←nlist     055444D ←prom4     043670D ←tloc
143136B ←erINT     043642D ←nloc      050470D ←promok    044400D ←tlst
143132B ←erRST     044310D ←nlst      037252T ←promtst   137104B ←tmp1
143140B ←erTRN     030204T ←nnlist    034256T ←putdsp1   062014B ←tmp2
143122B ←erWCR     043536D ←nnloc     026702T ←puttty1   137114B ←tmp3
144016B ←errcnt    044054D ←nnlst     026744T ←puttty2   136102B ←tpnorm
144006B ←errlok    030746T ←nnsrv     051146D ←ram1      062012B ←tref
050450D ←ex1loc    052062D ←noBAR     051062D ←ramcnt    062050B ←trefm1
050460D ←ex2loc    052136D ←noDBR     050714D ←ramer     032600T ←tsrv
143766B ←exck1     052402D ←noFCT     054552D ←ramin1    143276B ←ttibuf
143776B ←exck2     052526D ←noINT     054620D ←ramin2    143314B ←tticnt
043040T ←exit      052252D ←noRST     054706D ←ramin3    143310B ←ttiptri
143130B ←exrec     052642D ←noTRN     055012D ←ramin4    143312B ←ttiptro
043424D ←f10       052006D ←noWCR     055062D ←ramin5    143274B ←ttostat
043406D ←f9        051200D ←noram     055166D ←ramin6    022246T ←ttyin
060000B ←flag      051610D ←notar1    041016T ←ramins    046724D ←work
143732B ←flines    143266B ←np        040226T ←ramtst    132056B ←wrap
047774D ←flnloc    043062D ←np1       132072B ←rdtrsh    062056B ←x
062024B ←fstat     032254T ←nsrv      050766D ←rtceh     062054B ←xptr
022172T ←halt      062016B ←ntbkspa   046336D ←seelst    061004B ←ybuf1
042014T ←help      047674D ←octloc    025546T ←send      060004B ←ybuf2
047564D ←help1     144026B ←oldalfg   055540D ←ser1      132060B ←yincr
056762D ←hithlt    143602B ←oldcomn   056550D ←ser10     062004B ←yptr
057050D ←hitrep    062036B ←oldtref   055622D ←ser2      000006a asc
051266D ←hitret    143316B ←outstk1   055722D ←ser3      000006a asc
057134D ←hitrt     143440B ←outstk2   055770D ←ser4      177770a b
051316D ←hitrt2    023252T ←parse     056070D ←ser5      177770a b
057076D ←hitstr    062052B ←pcnt      056162D ←ser6      177770a b
056610D ←hittst    053116D ←pds1      056242D ←ser7      177770a b
144032B ←holdcha   053734D ←pds10     056344D ←ser8      177770a b
143632B ←holdcnt   053762D ←pds11     056446D ←ser9      177770a b
047524D ←home      053214D ←pds2      046576D ←sk21      177770a b
027006T ←initial   053326D ←pds3      047040D ←skip      020546T blavg
062006B ←intflg    053402D ←pds4      137120B ←smd       177760a bytecnt
041666T ←jrout     053502D ←pds5      143650B ←smdstr    177752a c
056700D ←jum       053530D ←pds6      143616B ←sml       177762a c
143142B ←junkln    053624D ←pds7      143610B ←smn       177770a c
046736D ←learn     053660D ←pds8      143664B ←smnstr    177754a c
047054D ←linela    053706D ←pds9      137112B ←smt       177742a c
047106D ←linelb    050004D ←pdsdis1   143660B ←smtstr    177770a c
```

```
177770a c         177766a i         177760a shift     032600t ~tsrv
177762a c         177764a i         177756a stackpt   022246t ~ttyin
177754a c         177770a i         020460t stinc
177766a c         177770a i         021270t targa
177744a carry     177756a i         020516t update
020772t chklst    177770a i         022106t wate
177762a ck1er     021274t init      020232t ybf2
177760a cks2er    022174a ircod.o   033734t ~addreal
020420t ckst      000000a ircod+in  041612t ~alarm
020252t cmpt      177770a j         024014t ~ascton
177770a cntr      177766a j         035200t ~atod
020104T cret      177764a j         041142t ~atodec
020266t crn       177770a j         035466t ~atodins
020066T csv       177764a k         034774t ~atodtst
043040a cuexit.o  177764a k         041460t ~atooct
177760a d         177762a k         043000t ~bds
177760a d         177766a k         042040t ~box
177752a d         177764a k         042740t ~bss
177766a d         020634t last      030440t ~c1list
177766a d         177752a lim       033252t ~c1srv
177764a d         022042t load      030406t ~c2list
177766a d         000004a loc       032774t ~c2srv
177766a d         000004a loc       030152t ~ddlist
177752a d         021334t loop1     030472t ~ddsrv
177770a d         021350t loop2     023452t ~decode
177754a d         020364t loopa     034530t ~display
020532t date      021014t loopb     030270t ~dlist
021146t decd      020640t loopc     042564t ~dmains
177756a dflg      021204t loopd     042230t ~dmats
177766a divide    177760a m         042170t ~dmatst
177762a divide    177764a m         032066t ~dsrv
177760a ercnt     106404a mtps      042014t ~help
177754a erflg     177762a n         027006t ~initial
177762a erflg     021230t nobis     041666t ~irout
177756a erloc     020272t nocrn     033530t ~lsrv
177766a excks1    000240a nop       022174t ~main
177764a excks2    020700t nord      030322t ~nlist
000001a exit      021010t norm      030204t ~nnlist
043056a fakcu.o   021054t ornd      030746t ~nnsrv
177766a fln       020740t noshf1    032254t ~nsrv
177750a frac      021114t noshf2    023252t ~parse
000000a halt      020576t nowrap    036176t ~pdsins
021302t here      000004a num       035562t ~pdstst
177770a i         000004a num       030236t ~pplist
177764a i         021214t pos       031354t ~ppsrv
177770a i         177746a powr      040122t ~prmins
177770a i         177750a ptin      037252t ~promtst
177770a i         177746a ptout     034256t ~putdspl
177770a i         000004a r1        026702t ~puttty1
177770a i         000003a r2        026744t ~puttty2
177766a i         000002a r3        041016t ~ramins
177770a i         000005a reset     040226t ~ramtst
177770a i         020752t retrn1    025546t ~send
177770a i         021126t retrn2    037066t ~termins
177766a i         000002a rti       036372t ~termtst
177764a i         020140t setflg    030354t ~tlist
```

DA
FILM

2–