

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 GOEDEL, W T WILSON, S C BASS

F30602-75-C-0082

UNCLASSIFIED

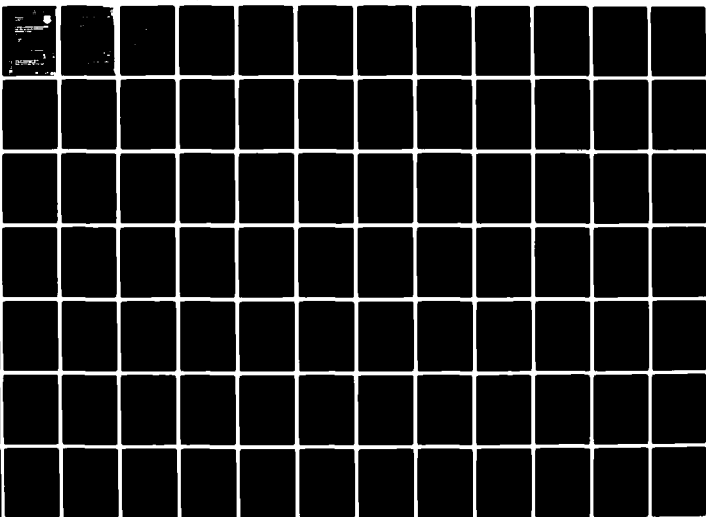
RADC-TR-80-308

NL

1 2

OF

10/10/80



AD A091630

[Signature]
[Illegible text]

[Signature]
[Illegible text]

[Signature]
[Illegible text]

[Illegible text block]

DISCLAIMER NOTICE

**THIS DOCUMENT IS BEST QUALITY
PRACTICABLE. THE COPY FURNISHED
TO DTIC CONTAINED A SIGNIFICANT
NUMBER OF PAGES WHICH DO NOT
REPRODUCE LEGIBLY.**

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER	
RADC TR-80-308 ✓	AD A091630		
4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED	6. PERFORMING ORG. REPORT NUMBER	
A MICRO-COMPUTER COMPUTATIONAL UNIT FOR AN IR-CCD INTRUSION DETECTION SYSTEM	Final Technical Report	N/A	
7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)	9. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
T.W./Goeddel W.T./Wilson S.C./Bass	F30602-75-C-0082 ✓	63714F 681ED0P1	
10. PERFORMING ORGANIZATION NAME AND ADDRESS	11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE	
Purdue University School of Electrical Engineering West Lafayette IN 47907	Deputy for Electronic Technology (ESE) Hanscom AFB MA 01731	October 1980	
13. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	14. NUMBER OF PAGES	15. SECURITY CLASS. (of this report)	
Same	184	UNCLASSIFIED	
16. DISTRIBUTION STATEMENT (of this Report)	17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)	18. SUPPLEMENTARY NOTES	
Approved for public release; distribution unlimited	Same	RADC Project Engineer: Lyn H. Skolnik (ESE)	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)	20. ABSTRACT (Continue on reverse side if necessary and identify by block number)	19a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
IRCCD Physical security Schottky array	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	N/A	

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

292000

fwe

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

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.



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

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 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 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.

Organization	NSA
Project	BISS
Task	Stare Time
Subtask	Stare Time
Phase	Phase 1
Version	1.0
Author	A. J. ...
Editor	A. J. ...
Reviewer	A. J. ...
Approver	A. J. ...
Date	23 Nov 73

TABLE OF CONTENTS

List of Figures.	v
List of Tables	vii
I. Introduction	
A. Purpose of Overall System.	1
B. Computational Unit	2
C. Start-Up Procedures for IR-CCD Intrusion Detection System	6
D. Description of Parameter Display Frame	8
E. Description of "Additional Features" Frame	16
F. Verification of System Operation	20
II. Hardware	
A. Overview of Hardware Operation	25
B. Step-by-Step Discussion of IR-CCD Interface Operation.	27
C. Signal Pin Assignments	31
III. Software	
A. Overview of Software Operations.	37
B. Implementation of Software	44
C. Memory Organization.	84
IV. Maintenance	
A. Preliminaries to Testing	88
B. Calibration Procedures and Use of Maintenance Software	88
C. PROMS.	102

V. References

Appendix A. Parts List, Manufactures and Vendors

Appendix B. Software Listings and Symbol Table

LIST OF FIGURES

<u>Figure</u>	<u>Title</u>	<u>Page</u>
1.1	System level depiction of the computational unit described in this report together with its connections with the IR-CCD detector system	3
1.2	Flow diagram of the major calculations involved in a single cycle of the computational unit software . . .	5
1.3	Initial parameter display frame	9
1.4	Frame showing typical prompt for changing a parameter . .	11
1.5	Flow diagram of parameter charging operation.	14
1.6	"Additional features" frame	17
1.7	Flow diagram illustrating additional feature selection process	21
1.8	Setup used to verify system operation	23
2.1	Functional level drawing of IR-CCD interface board. . . .	26
2.2	Timing diagram of IR-CCD interface operation.	28
2.3	Pin assignment of 40-pin "berg" connector on IR-CCD interface board	33
2.4	Pin assignments of 40-pin "berg" connectors on DMA board	34
2.5	Pin assignment of 16-pin dip connector to IR-CCD device .	35
3.1	Flow diagram of main executive routine.	38
3.2	Flow diagram of DMA "interrupt service" routine	39
3.3	Timing of stare time changing sequence.	42
3.4	Structure of output queues.	48
3.5	Layout of "x" vector.	65

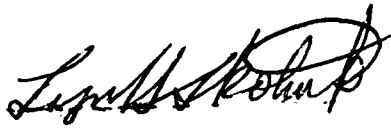
3.6	Layout of "m" vector.	66
3.7	Layout of "a" vector.	68
3.8	Computation of background spatial average and stare time changing operation	70
3.9	Uses of stare time information.	73
3.10	Distribution of calculations over P packets.	77
3.11	Computation of background time average and corresponding decision threshold.	80
3.12	Target/No target decision process.	82
3.13	Partitioning of memory between RAM and PROM.	85
3.14	Locations of several important arrays in RAM	86
4.1	Board locations in card cage	89
4.2	Wiring diagram of IR-CCD custom interface.	90
4.3	Typical view of terminal screen during either monitor or workspace portions of terminal/serial interface test.	101
4.4	Location of PROMs on boards, ...	103

LIST OF TABLES

<u>Table</u>	<u>Title</u>	<u>Page</u>
1.1	Possible parameter values and their defaults	13
1.2	Brief descriptions of software maintenance routines. . . .	19
3.1	Special ascii characters and their corresponding messages.	50
4.1	Example of output from PDS logic test.	95
4.2	Possible messages displayed by DMA test.	97
4.3	Machine language code for performing checksums on individual PROMs	.104
4.4	Correct results for checksum test of Table 4.3	.105

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.



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 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.

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 one of four 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

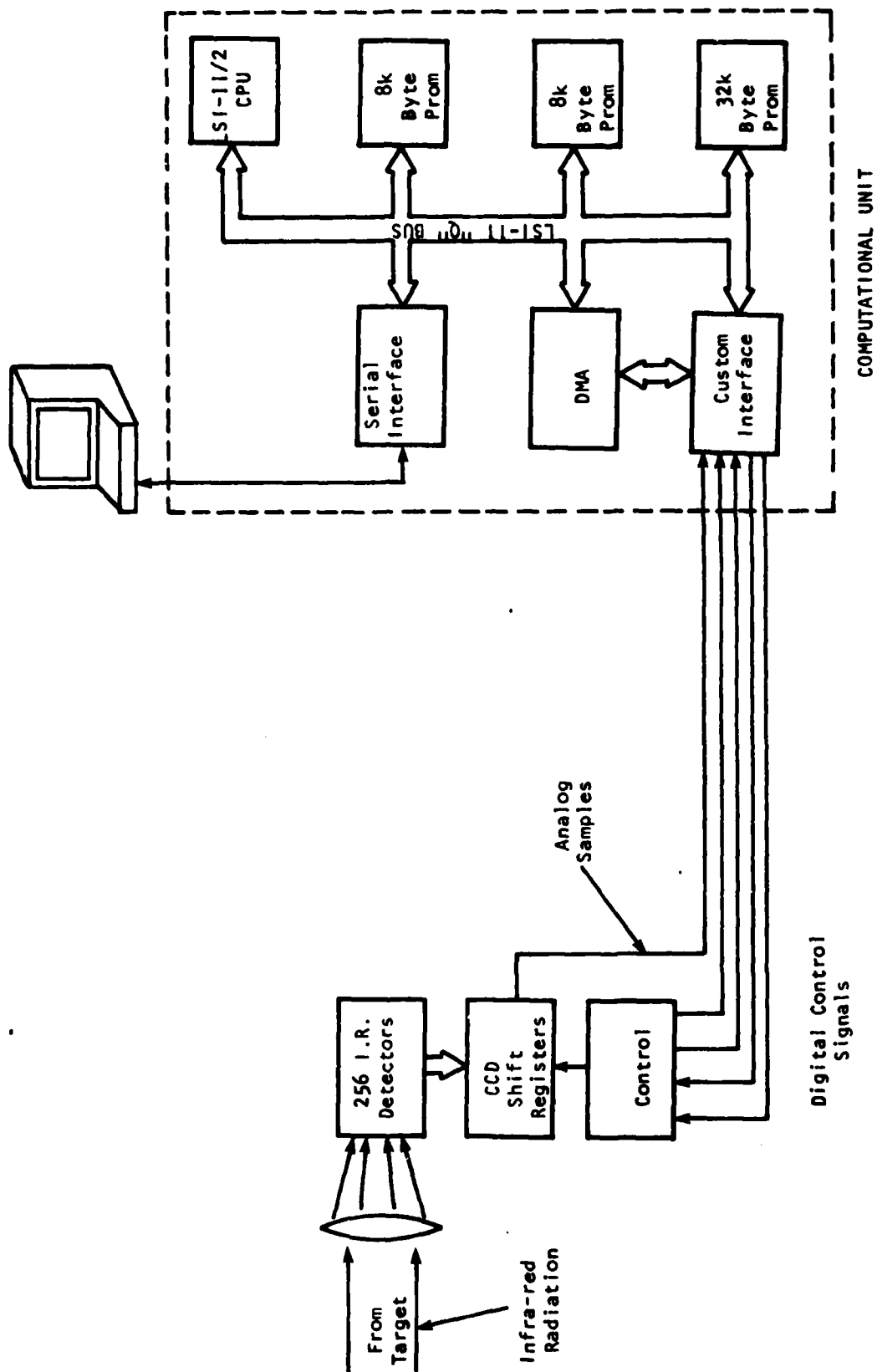


Figure 1.1 System level depiction of the computational unit described in this report together with its connections with the IR-CCD detector system.

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.

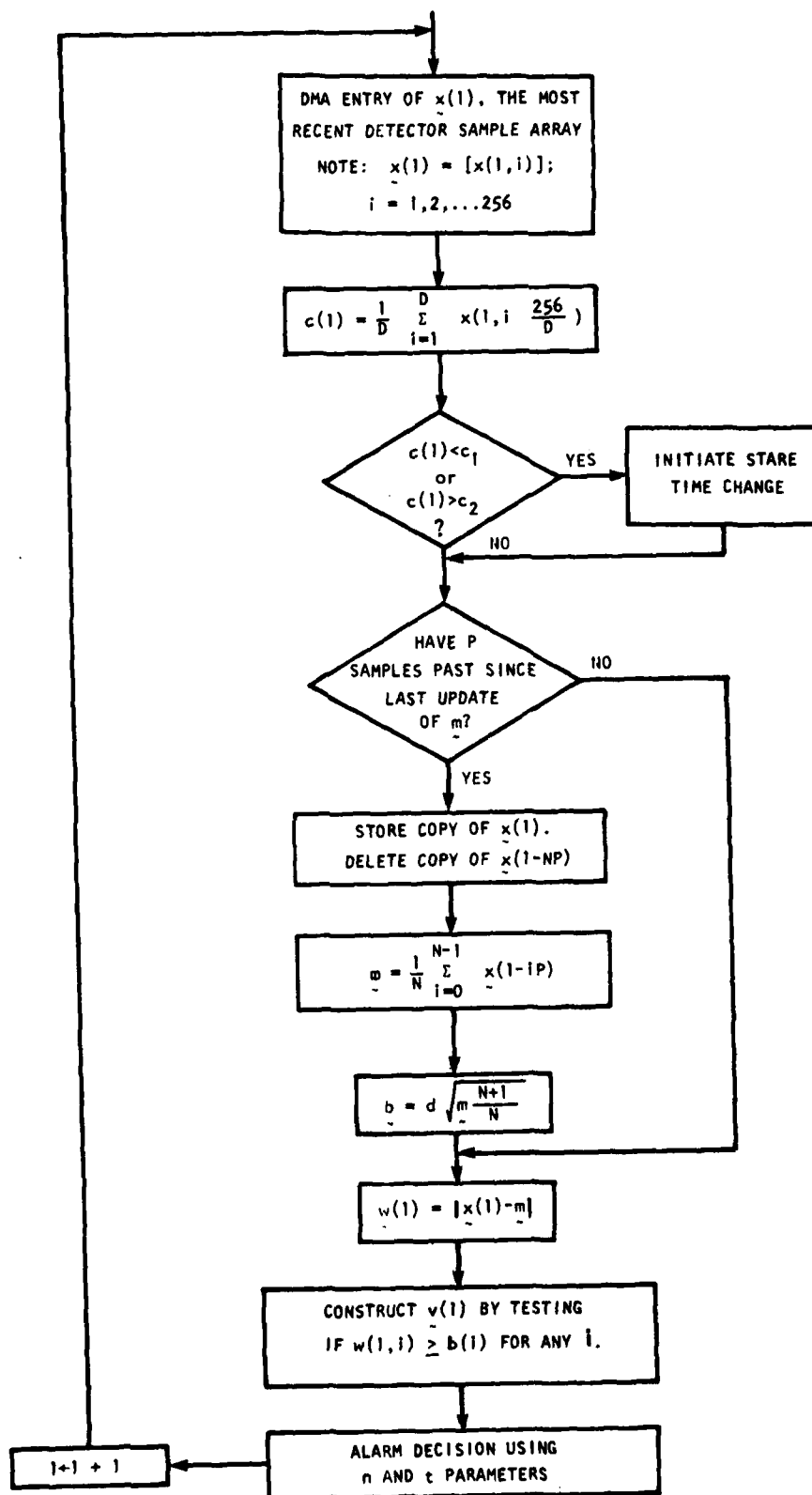


Fig. 1.2 Flow diagram of the major calculations involved in a single cycle of the computational unit software.

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 "a" symbol. Whenever it is desired to begin execution of the IR-CCD Intrusion Detection program, the command "200006" 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 "200006" command after receiving the "a" 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 "200006" 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 Tektronix 4024 terminal, this is accomplished by hitting the "BREAK" key rapidly twice in succession. The processor should return to ODT

mode with the "a" prompt appearing on the screen.

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

- (1) Connect Tektronix 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 (a) 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 $2 \leq N \leq 40$. N=?

Figure 1.4 Frame showing typical prompt for changing a parameter.

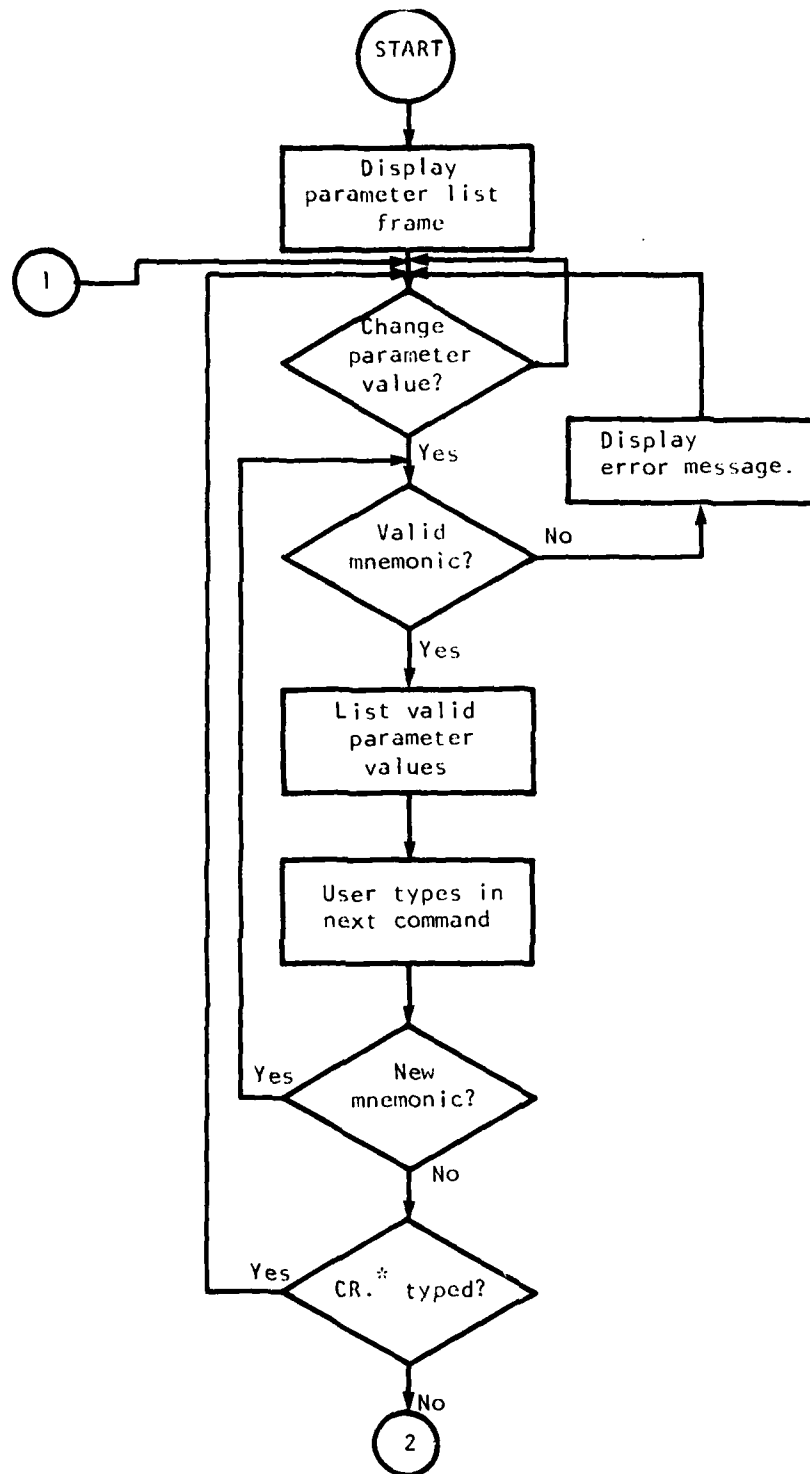
messages at the bottom of the screen are restored to their original state, informing the user of how to change a parameter value. All allowed parameter values and their default values are listed in Table 1.1. Figure 1.5 shows a flow chart illustrating the entire parameter changing operation.

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 values 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 $N \cdot P$ packets into the past. Thus the alarm is disabled for $N \cdot P$ 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.

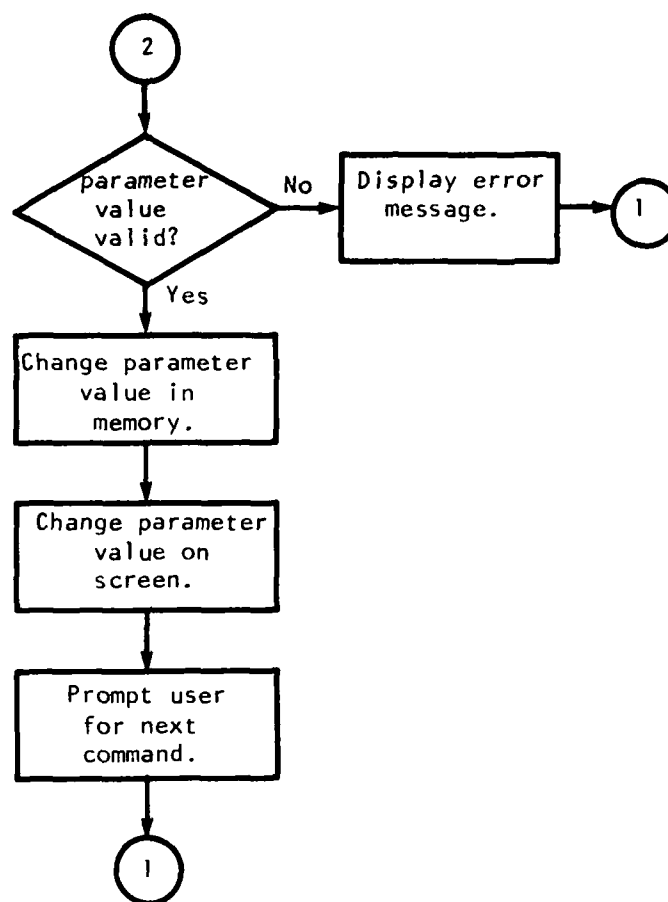


Figure 1.5 continued.

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.

<u>Test</u>	<u>Description</u>
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 Tektronix 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

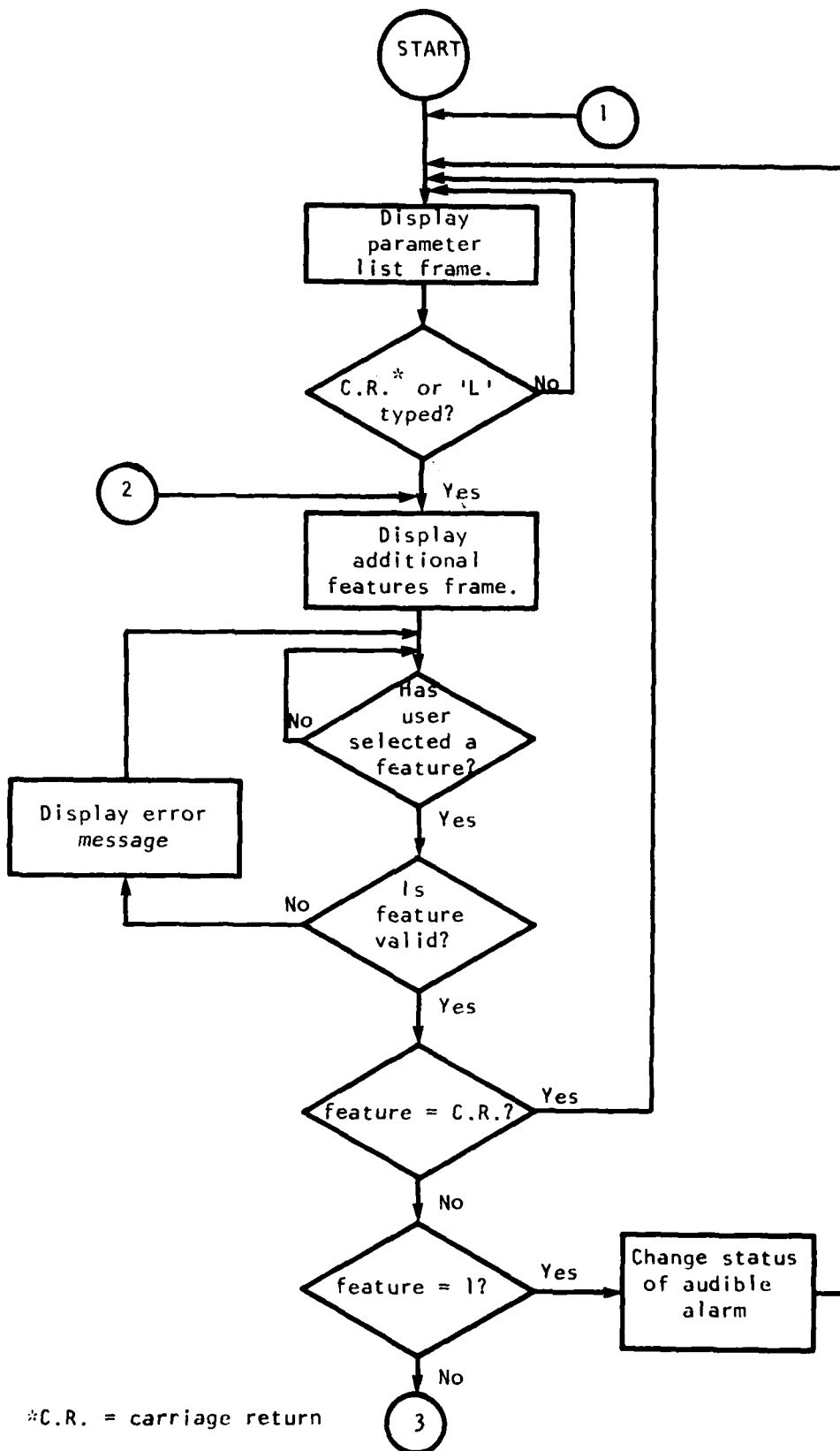


Figure 1.7 Flow diagram illustrating additional feature selection process.

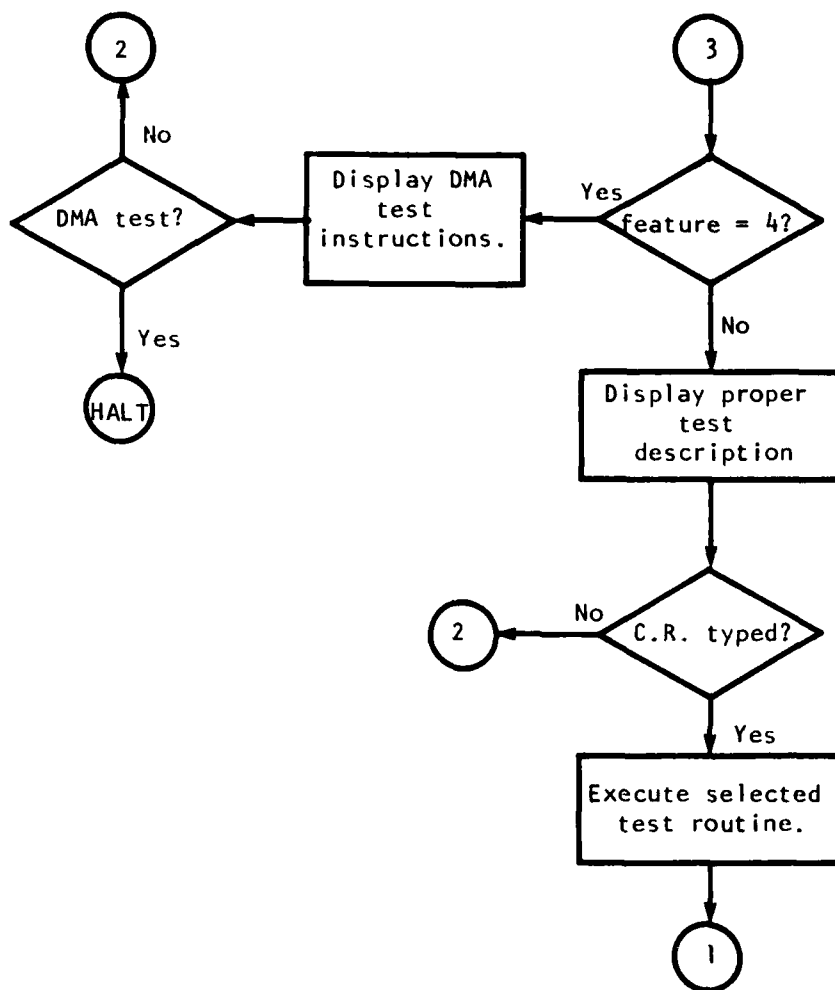


Figure 1.7 continued.

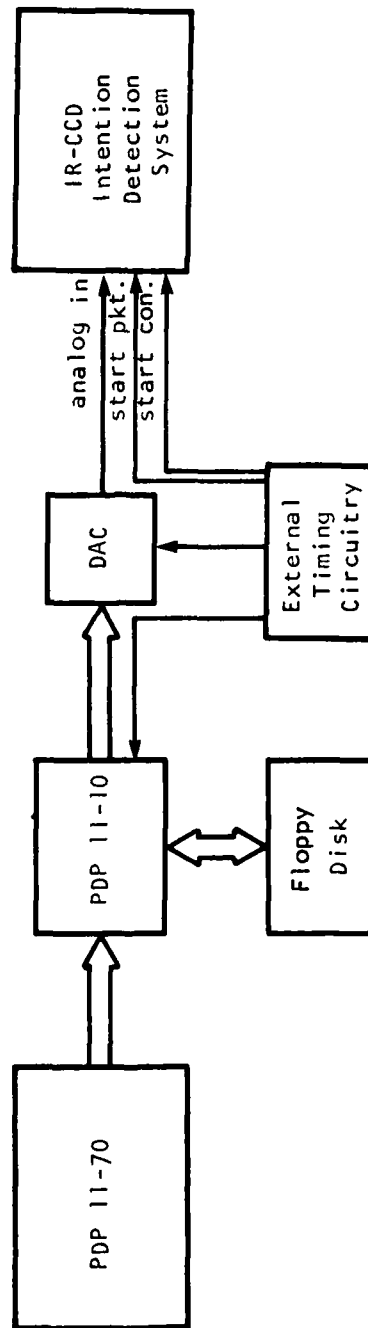


Figure 1.8 Setup used to verify system operation.

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 aperture 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

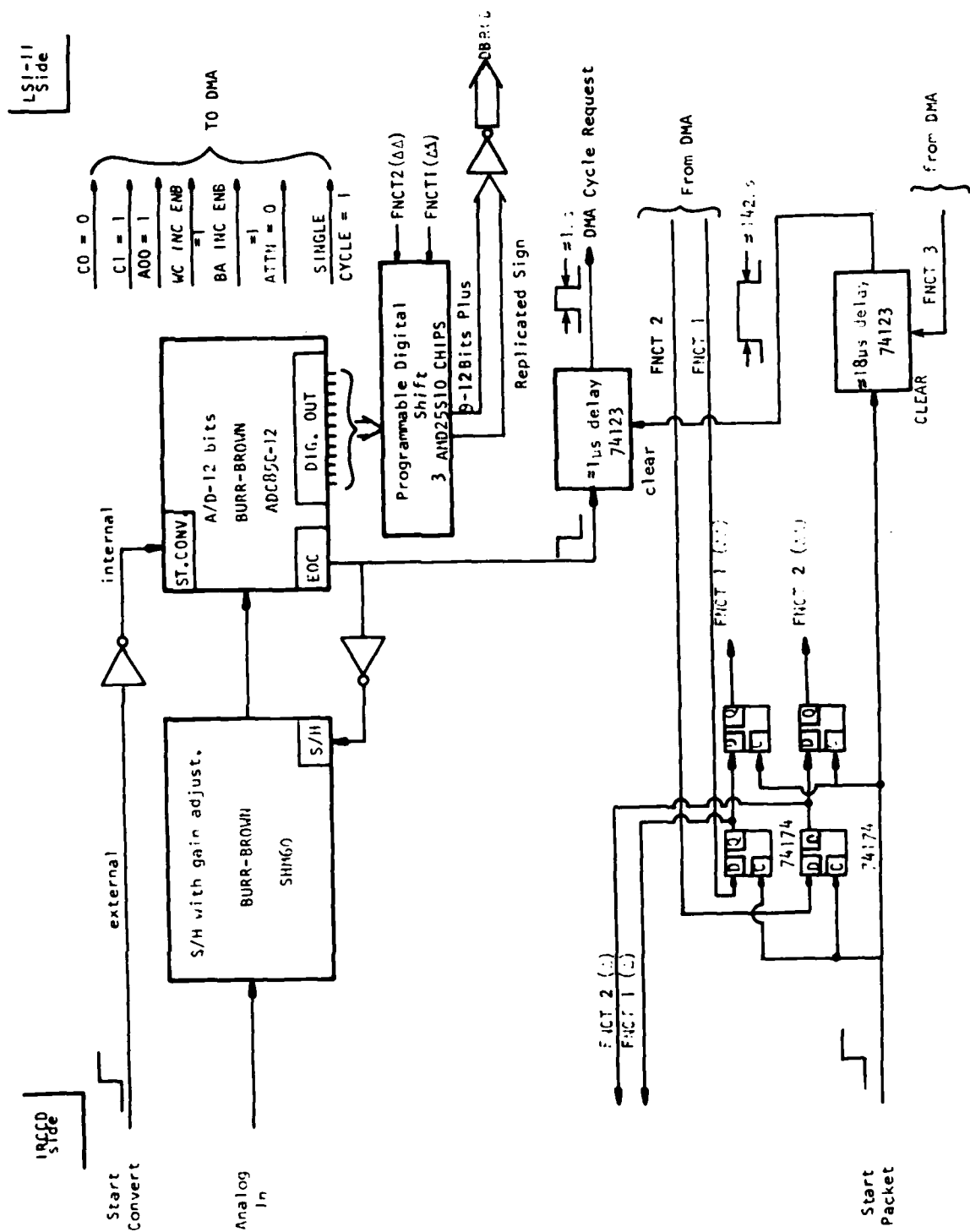


Figure 2.1 Functional level drawing of IR-CCD interface board.

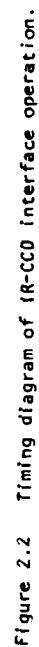
LSI-11/03 bus is converted to ± 15 volts by this device. Specifications for these three devices are given in Appendix Section A.

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 packet 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.



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-flops 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($\Delta\Delta$) and function 2($\Delta\Delta$). 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 μ sec. pulse. The rising edge of this negative-going pulse triggers another one-shot producing a 1 μ sec. positive-going pulse. The rising edge of this 1 μ sec. positive-going pulse in turn triggers yet another one-shot (a high valued function 3 tied to its clear) producing a $\approx$ 142 μ sec. positive-going pulse. The result of all this is the production of a $\approx$ 142 μ sec. positive-going pulse that is delayed 18 μ sec. 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 μ sec. pulse for EOC. Ten μ sec. corresponds to the specified time required by the ADC85C-12 to do a conversion to binary representation of an analog voltage. Eight μ sec. 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 hold an analog voltage, and the falling edge to signify end of conversion.

The falling edge of EOC triggers a one-shot producing a 1 μ sec. negative-going pulse. The rising edge of this negative-going pulse triggers another one-shot producing a 1 μ sec. positive-going pulse. It is this positive-going 1 μ sec. pulse that acts as the input to the clearable DMA one-shot. The rising edge of this 1 μ sec. positive-going pulse triggers the DMA one-shot (only if clear is high) producing positive-going 1 μ sec. 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 momentarily. Dropping function 3 low will cause the ≈ 142 μ sec. 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 not critical since DMA Cycle Requests after sample #255 are ignored by the DMA until after function 3 is momentarily set low. 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 $\approx 9 \mu\text{sec.}$ after each internal start convert falling edge ($\approx 8 \mu\text{sec.}$ for conversion plus $1 \mu\text{sec.}$ 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)

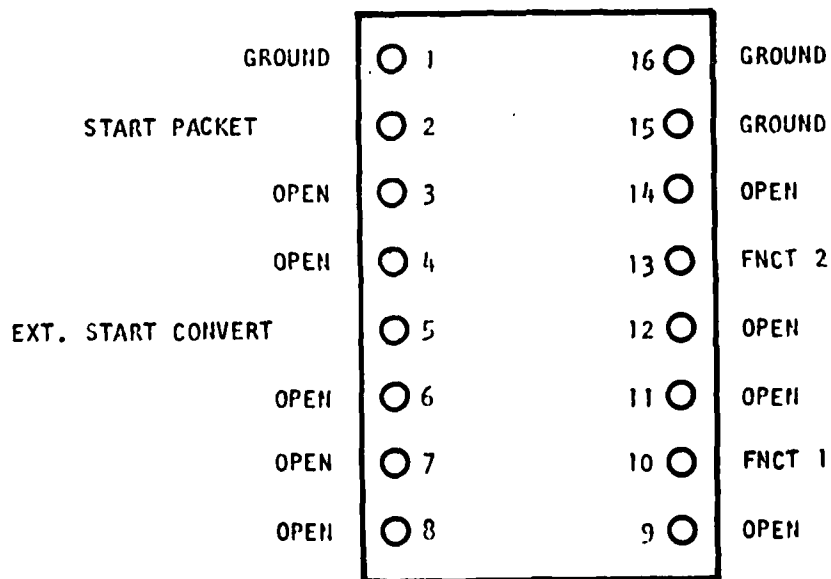
Figure 2.3 Pin assignment of 40-pin "berg" connector on IR-CCD interface board.

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.



16 PIN IC SOCKET
TOP VIEW
AT IRCCD DEVICE

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 transferred from the main executive routine (or some subroutine that was ul-

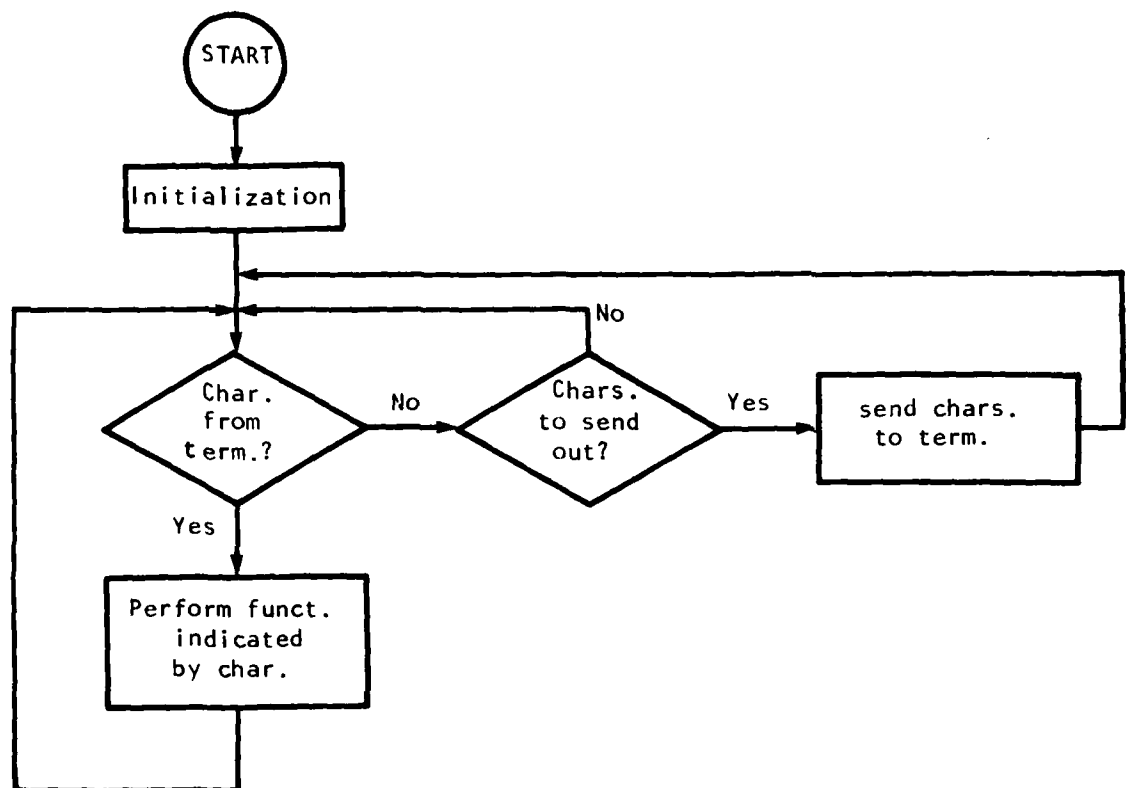


Figure 3.1 Flow diagram of main executive routine.

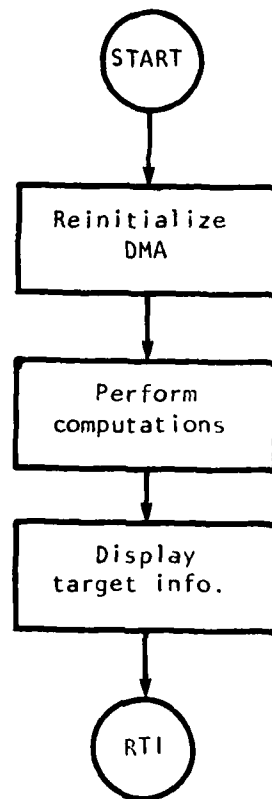


Figure 3.2 Flow diagram of DMA "interrupt service" routine.

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.

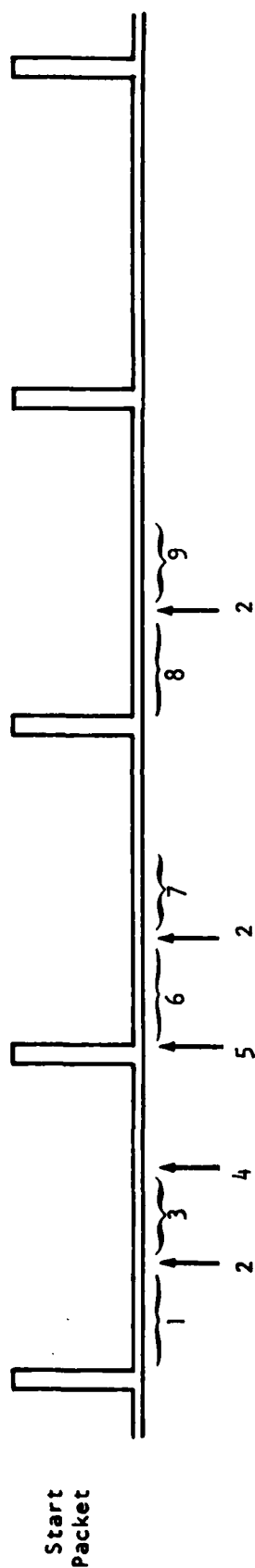
The DMA must now be reinitialized so that it is ready to start accepting the next packet of 256 data samples. This initialization is accomplished in several steps. First of all, the DMA Word Count Register (WCR) must be set to -256. This allows the DMA to perform 256 data transfers into memory for the 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 transferred 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 second packet after 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 store 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 store time change.
8. Packet #n + 2 being loaded by DMA into memory. This is the first packet which uses the new store time and hence is shifted accordingly by the interface electronics.
9. Computations check for store 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", "putty1", "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

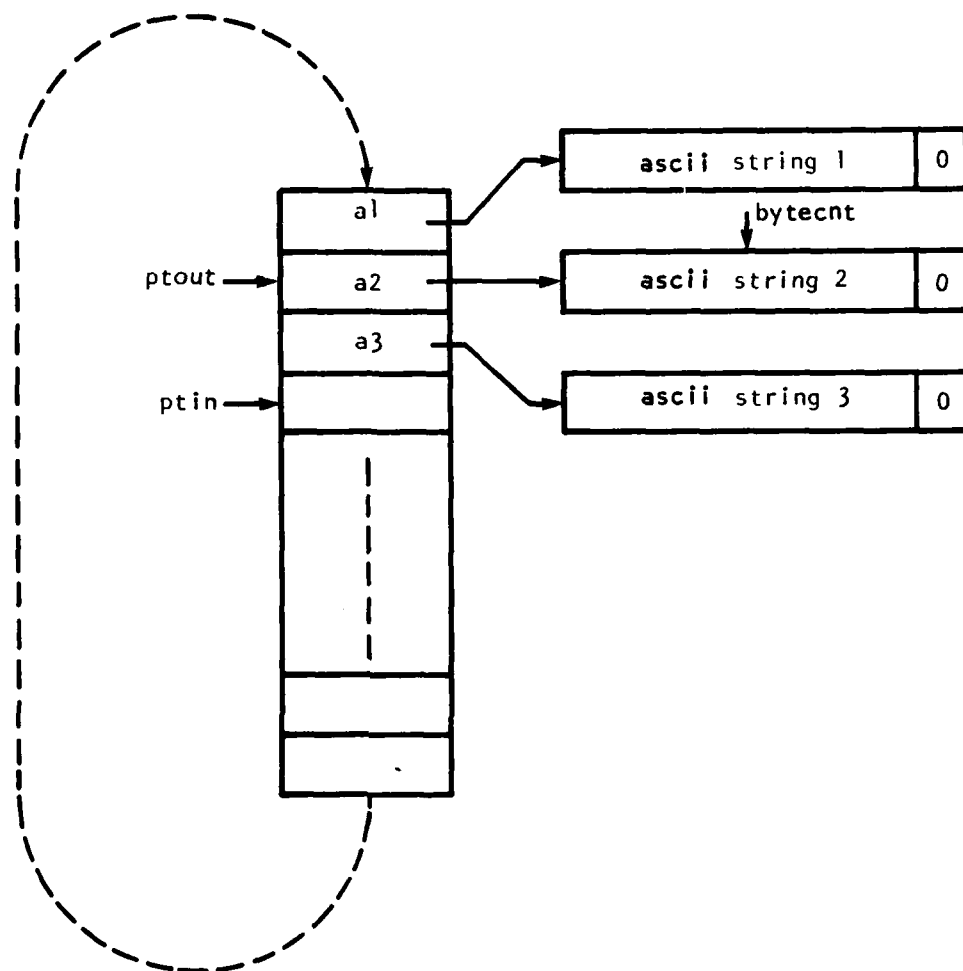
character data from the terminal. Called from the main executive routine whenever a character is present at the input of the serial interface, this routine first places characters into a temporary buffer until an end of command character is received (carriage return). As each character is received, it 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 is 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 formatted 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 transferred 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 "asc-

ton" routine is transferred 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 except 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 "putty1" 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) termstst - 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 accumulator 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 accumulated 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 or 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".

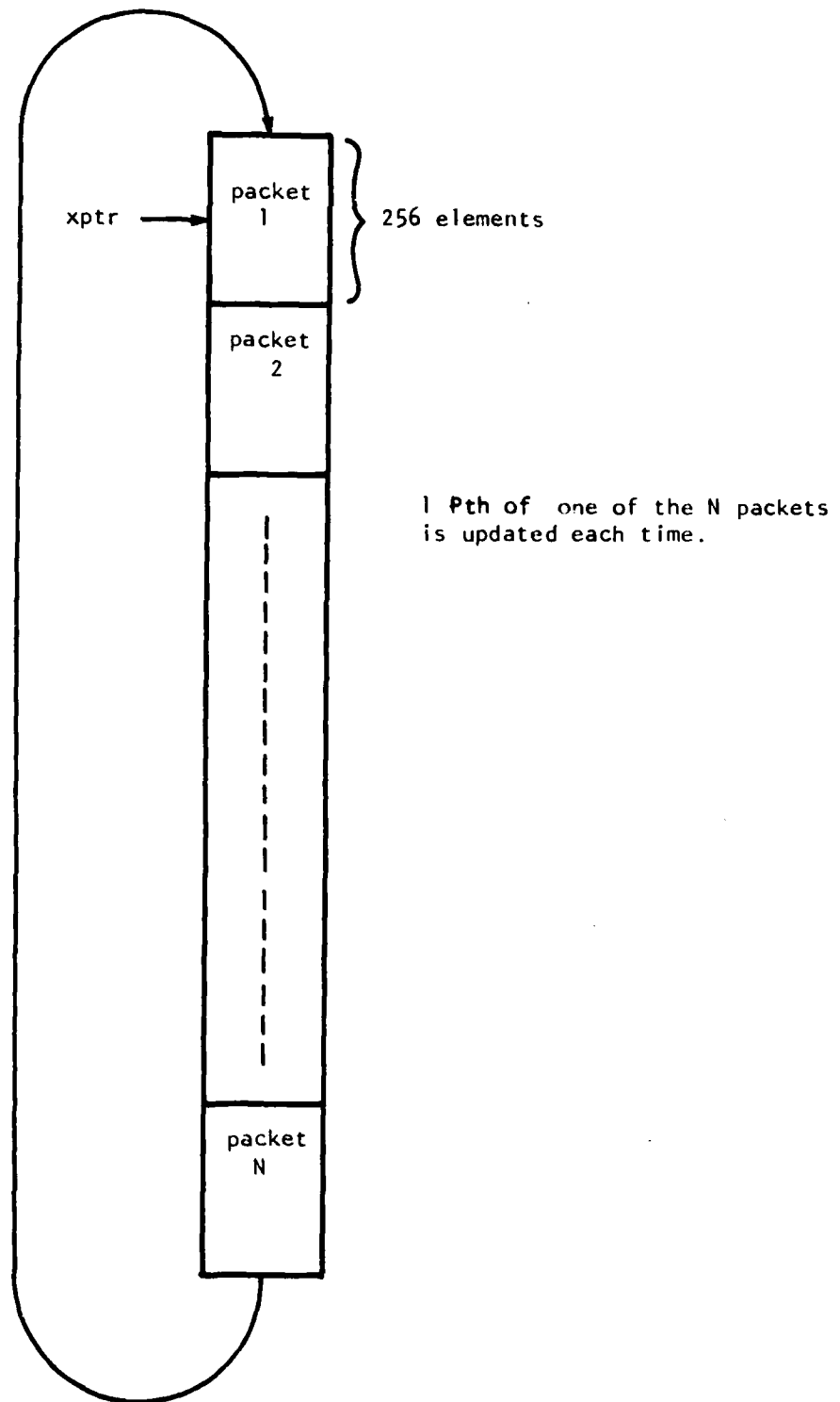
(53) dmaint - This is a DMA interrupt service routine that is entered during the DMA diagnostic test. The only function it serves is to set a flag indicating that the DMA interrupt service request has been honored.

(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 $1/P^{th}$ 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.

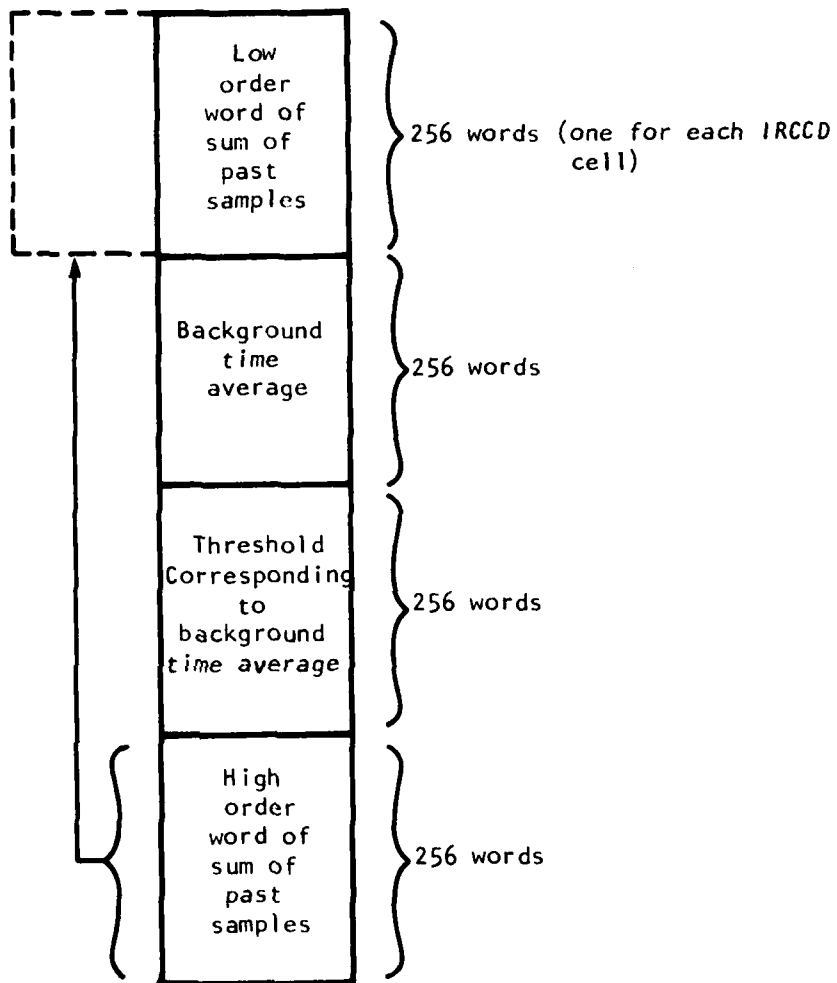


Figure 3.6 Layout of "m" vector.

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:

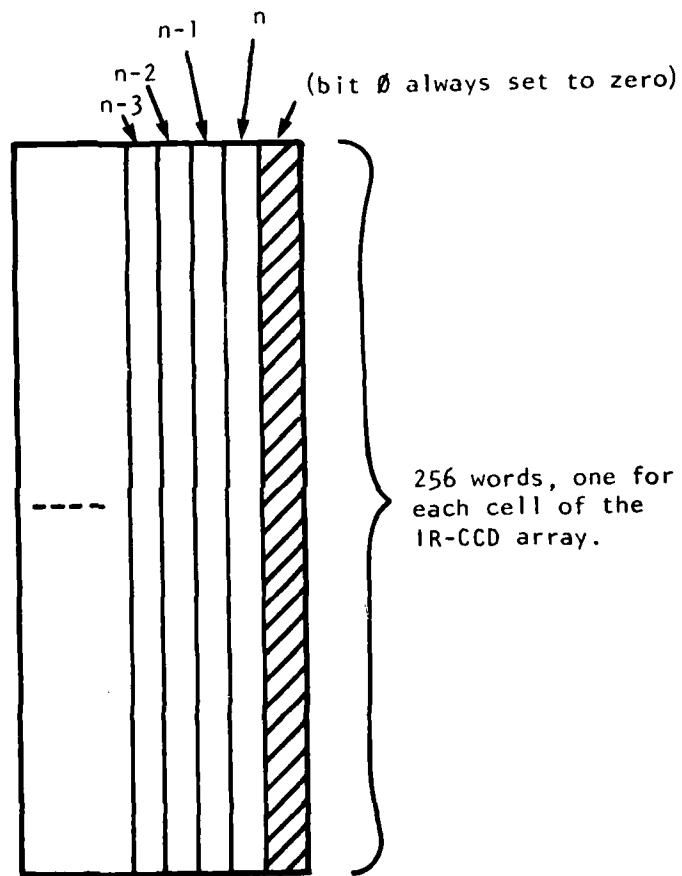


Figure 3.7 Layout of "a" vector.

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 D^{th} element in this array, "yptr" plus $2 * \text{"tbgd"}$ points to the $2 * D^{\text{th}}$ element in the array, etc. The section of code that computes the sum of D terms uses this procedure for extracting every D^{th} 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 store 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 store 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".

Store 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 store time increase is in order (C1). These thresholds are not constant for different store times. When operating at tref, the incoming samples are roughly in the range 256 to 511. When the

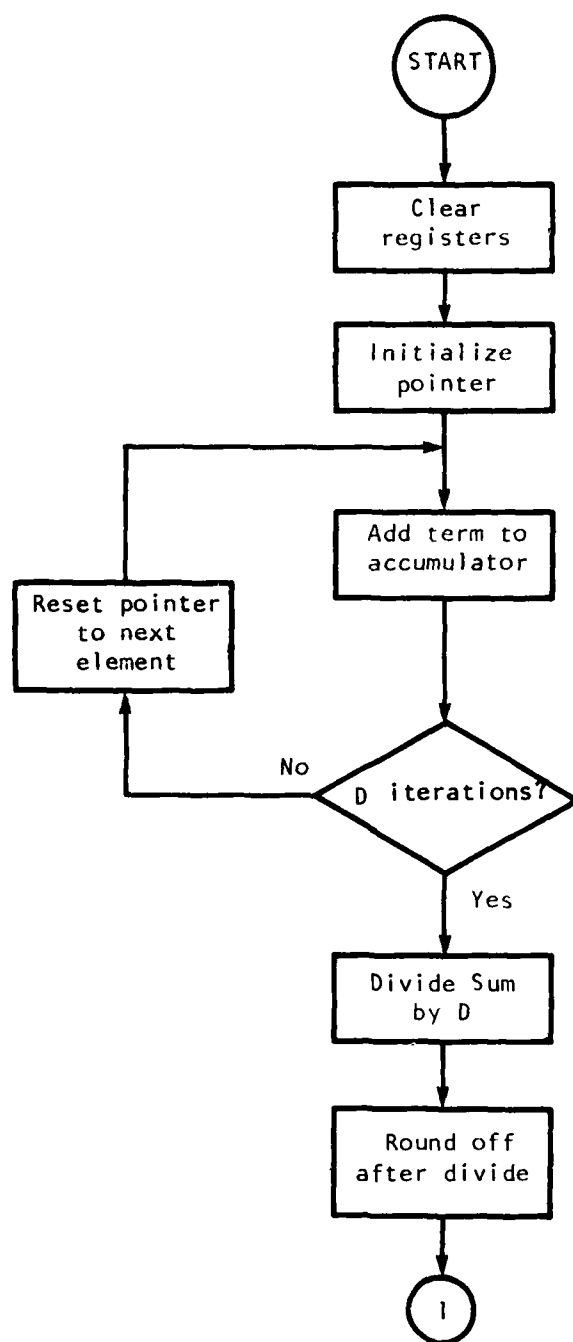
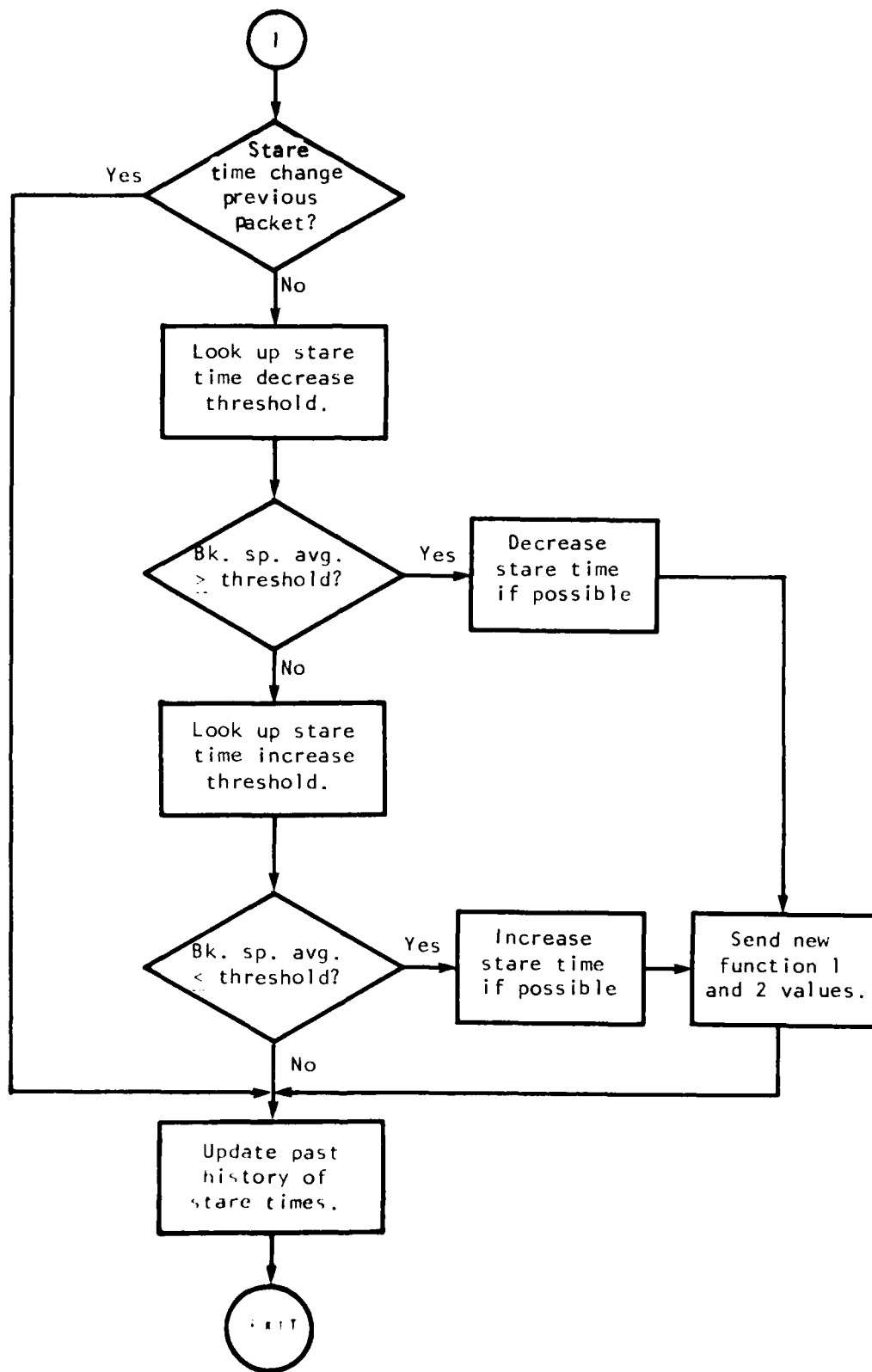


Figure 3.8 Computation of background spatial average and store time changing operation.

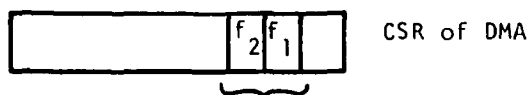
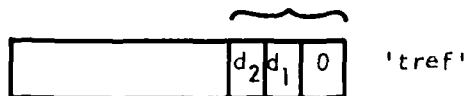


stare time decreases to $tref/2$, the incoming samples are roughly in the range 512 to 1023. Similarly, at $tref/4$ the samples range from 1024 to 2047, and at $tref/8$ from 2048 to 4095. Therefore, as $tref$ decreases, the thresholds must be increased by a factor of two at each change. Similarly as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



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

Start of Array 'C1':

8 * C1
(Starting address+2) 4 * C1
(Starting address+4) 2 * C1
(Starting address+6) C1

Start of Array 'C2':

8 * C2
4 * C2
2 * C2
C2

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 transferred 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 transferred 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".

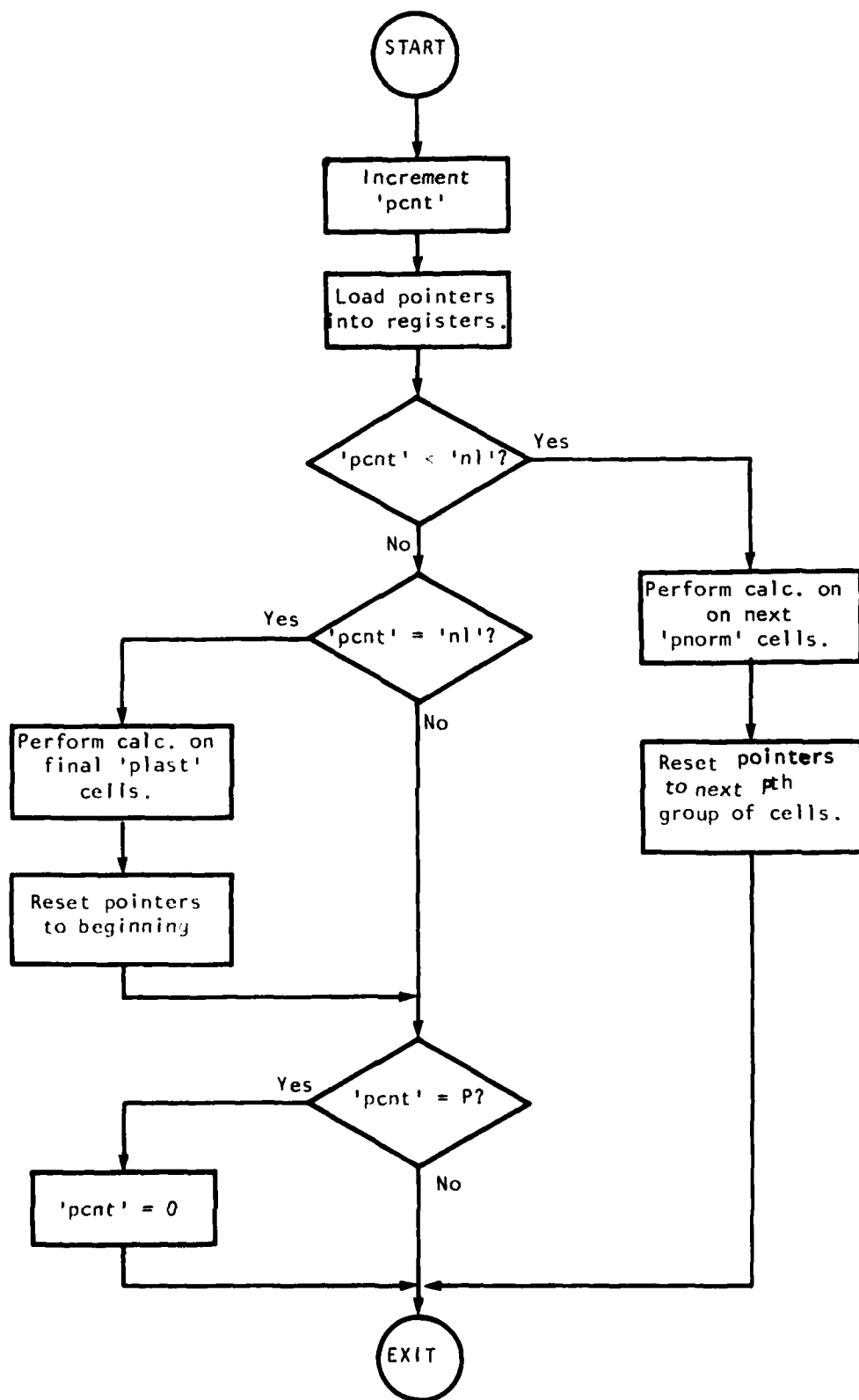


Figure 3.10 Distribution of calculations over P packets.

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 iteration 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 contained 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.

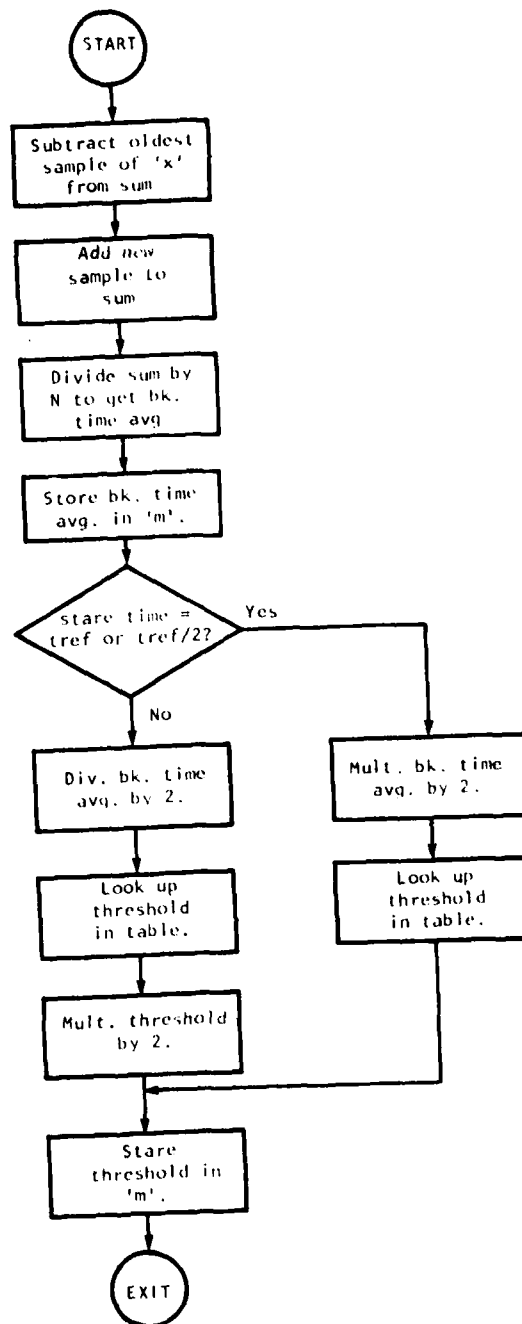


Figure 3.11 Computation of background time average and corresponding decision threshold.

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 been 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.

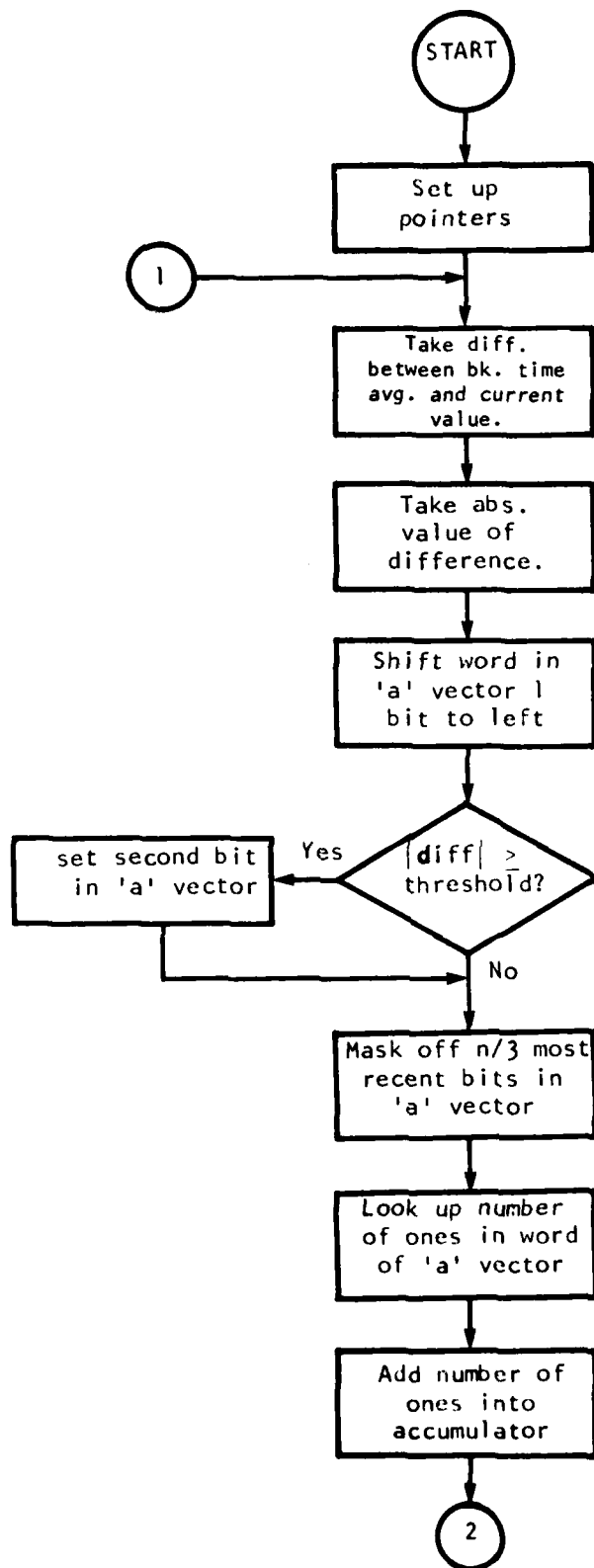


Figure 3.12 Target/No target decision process.

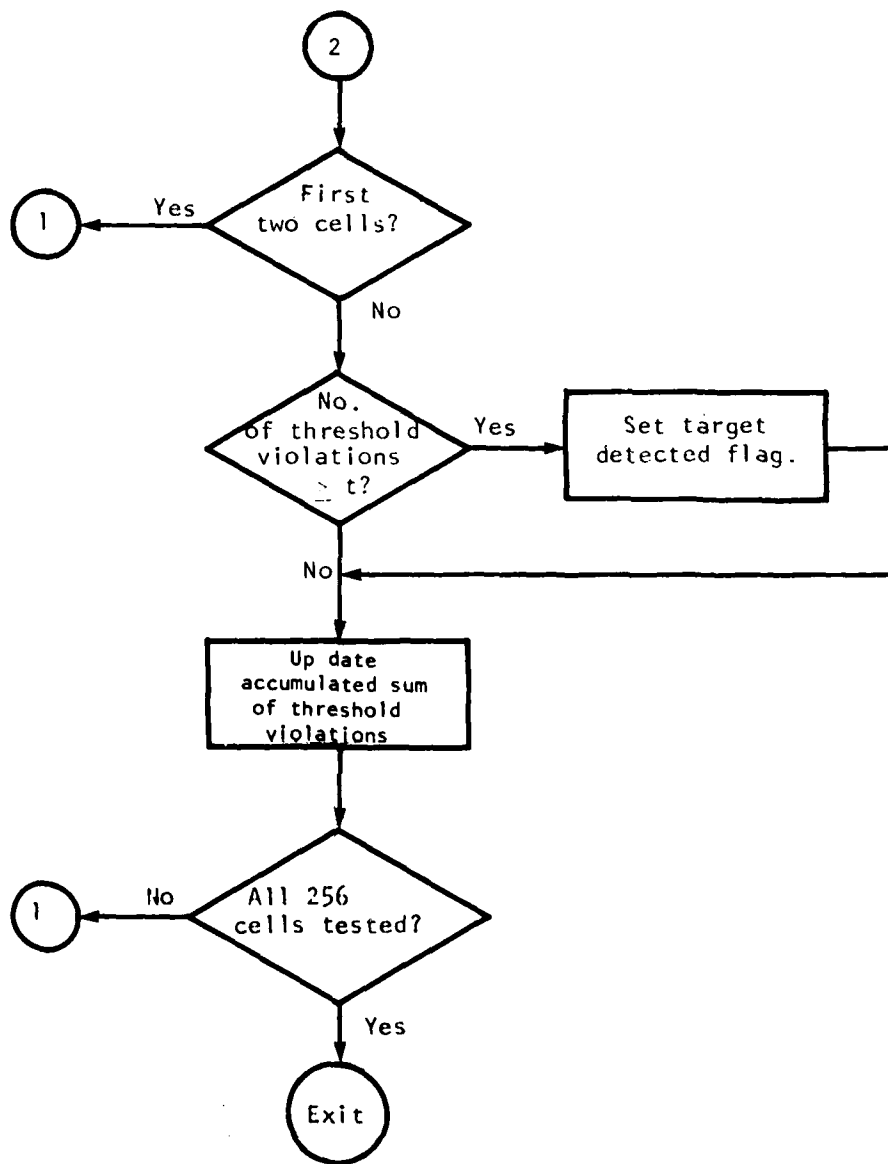


Figure 3.12 continued.

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 board 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-

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 GOEDEL, W T WILSON, S C BASS F30602-75-C-0082
RAC-TR-80-308 NL

RADC-TR-80-308

NL

4.2

END
DATE
FILMED
12 80
DTIC

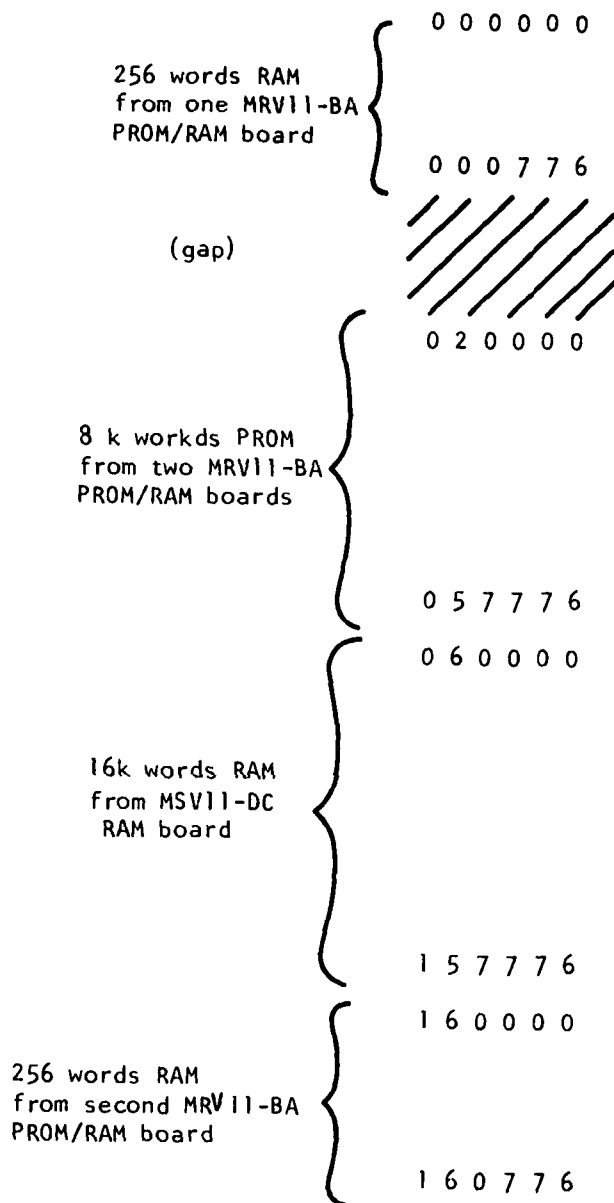


Figure 3.13 Partitioning of memory between RAM and PROM.

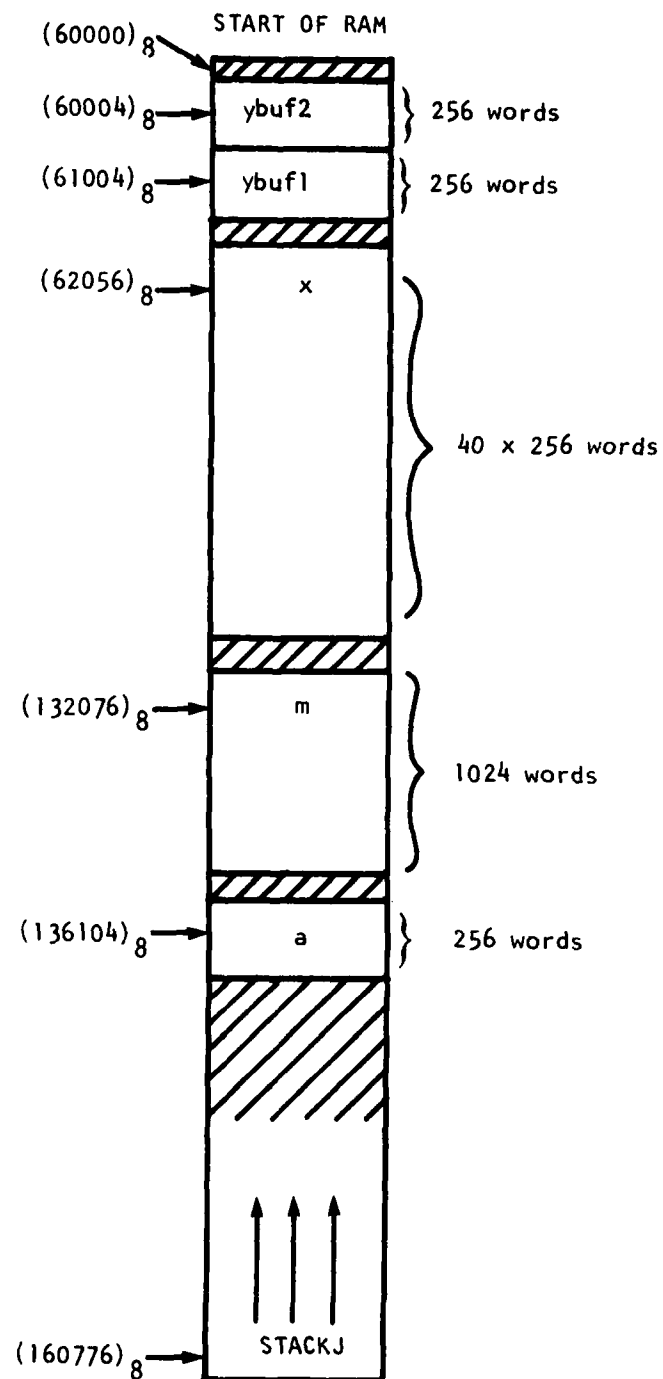


Figure 3.14 Locations of several important arrays in RAM.

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.

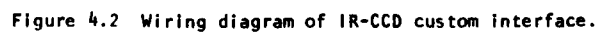
To use the card extender(s), shut down power as above, remove board to be inspected, insert card extender with finger "notch" to the left. Insert board into card extender (you should provide support to the rear of the board). Power up as above. When the interface board is to be inspected, removal of the DMA board below the interface is required due to the thickness of the interface board. After DMA and Interface removal, the DMA should be inserted for proper testing to occur. You do not 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.

KD11-HA LSI-11/2 Processor Board	DCK11-AC IRCCD Interface Board
DRV11-B DMA Interface Board	
MSV11-DC 16K word RAM Board	DLV11 Serial Line Unit
MRV11-BA UV PROM/RAM Board lower 4K words	MRV11-BA UV PROM/RAM Board upper 4K words

Figure 4.1 Board locations in card cage.



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 con-

nected 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 possibilities 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 use 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 accumulated 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 discrepancies 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.

Figure 4.3 Typical view of terminal screen during either monitor or workspace 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 bytes 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 "600006" 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

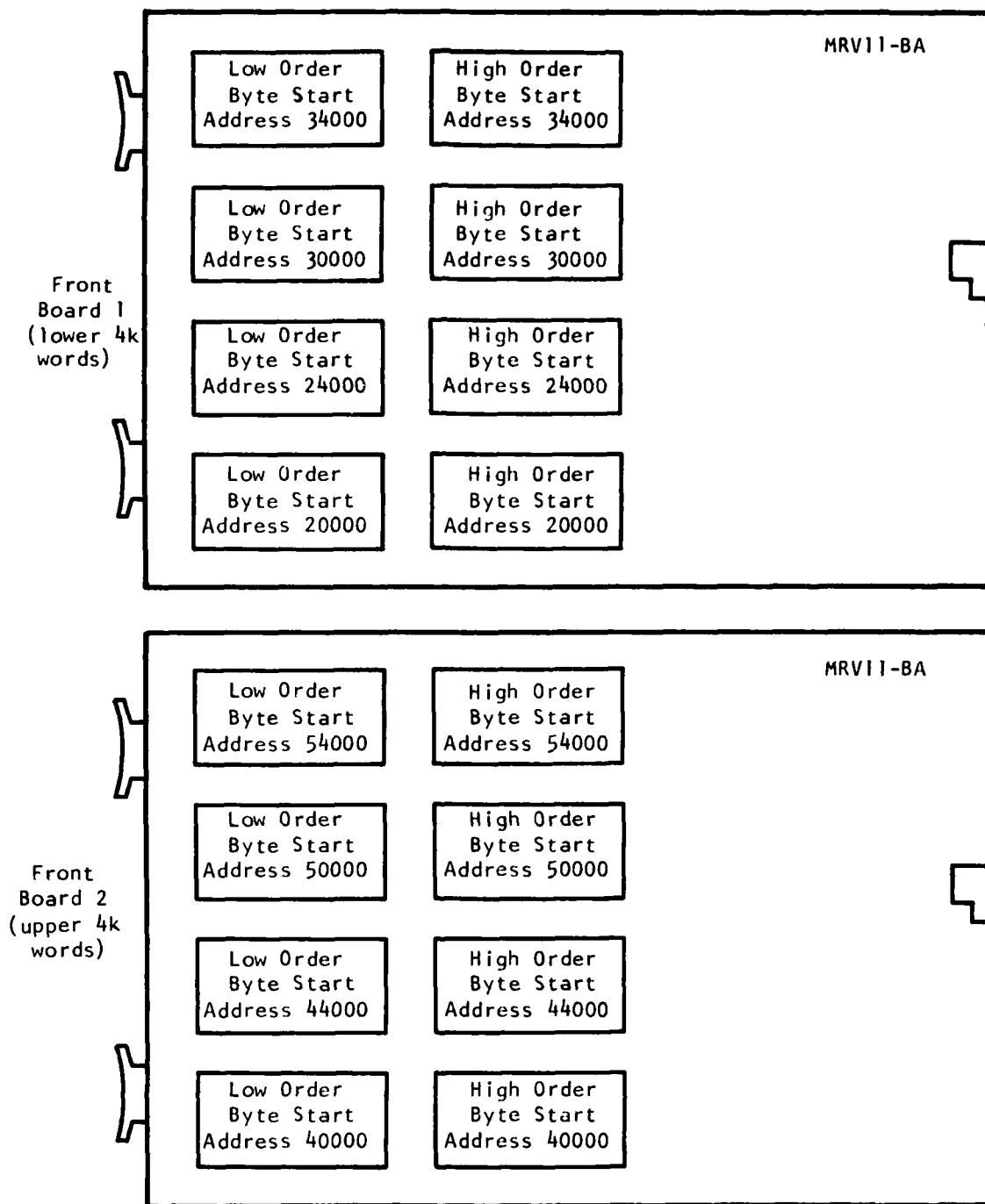


Figure 4.4 Location of PROMs on boards (top view).

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
65032	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-0P-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 $\pm 15v$ 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 TIRBUF 0177562 /* address of serial interface input buffer */
#define TTYSTAT 0177560 /* address of serial interface input status word */
#define TTYSTAT 0177564 /* address of serial interface output status word */
#define TTORBUF 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 ENDL4K 037776 /* address of end of lower 4k words of PROM */
#define ENDU4K 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 bufin[82] /* buffer to hold line of chars in ttr 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 *pvec /* pointer to past data vector */
int pvec[1024] /* past data vector */
int m[1024] /* sum of past terms: bkgnd time avg. thresholds */
int al[256] /* history of threshold violations */
int blab[1024] /* table to hold target thresholds */
int *mptr /* pointer to m vector */
int *aptr /* pointer to a vector */
int *bptr /* pointer to blab */
int pmt /* ptr to indicate which fth of calcs to be done */
int ntbl /* number of terms in background spatial avg */
int ltbl /* two times D */
int dchft /* no. of bits to shift to divide by D */
int fltat /* status of previous packet store time changing */
int ccl[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      initia      /* flag to tell if calcs should be skipped */
int      trer        /* state time for current packet */
int      trerml      /* state time one packet into is 1 */
int      oldtrer     /* state time for two packets into past */
int      wrap        /* address to wrap around circular queue in x */
int      sincr       /* counter used to point to location in current input vector */
int      mask        /* mask to look at past and most recent threshold violation bits */
int      rnorm       /* number of oval the first bit raised */
int      lenorm      /* two times rnorm */
int      rl          /* number of r/r calculations on planned time avg are performed */
int      rdtrsh      /* value used for rounding off after divide */
int      elast       /* no. of cells updated the nl pass */
int      tmr1        /* place for temporarily storing register value */
int      tmr2        /* place for temporarily storing register value */
int      tmr3        /* place for temporarily storing register value */
int      can         /* equal to eqpt/(N+1)/N */
int      dmscr       /* used to store DMA CSR values */
char      *stlncat    /* pointer to serial interface input status register */
char      *stloutat   /* pointer to serial interface output status register */
char      titbuff[10] /* input char buffer */
char      *tittrib    /* pointer to input char buffer for placing chars in buf */
char      *titrtrob    /* pointer to input char buffer for removing chars from buf */
int      titcnt       /* 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      *bistrl     /* pointer to elements pointed to by elements of outstk1 */
char      *bistr2     /* pointer to elements pointed to by elements of outstk2 */
char      *bistrl1    /* pointer to pointer for removing elements from output queue 1 */
char      *bistr21    /* pointer to pointer for removing elements from output queue 2 */
int      bton1       /* value indicates which byte of string pointed to by elements of outstk1 */
int      bton2       /* value indicates which byte of string pointed to by elements of outstk2 */
int      datares     /* temporary location for storing new parameter values */
int      command      /* holds number corresponding to current command */
int      oldcommand   /* holds number representing previous command entry */
int      audalarm     /* flag indicating whether or not audible alarm is enabled */

/* Locations for storing parameter values */

int      bnd, b1, b2, b3, md, sm, ml, cl, c2, sm1

char      datstr[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      b1sstr[4],
          b2sstr[4],
          b3sstr[4],
          mdstr[3],
          sm1str[4],
          sm2str[4],
          clstr[5],
          c1str[5]

char      aldbin[11]  /* buffer for holding ascii representation of octal number */
char      aldbin[12]  /* buffer for holding ascii representation of binary number */
char      flines[6]   /* buffer for storing ascii representation of function lines */
int      alodbyt      /* location for storing word from A to D converter */
int      chksum1      /* checksum for lower 4K of PROM */
int      chksum2      /* checksum for upper 4K of PROM */

```

```

char      chk1[7]; /* buffer for storing ascii representation of chksum1 */
char      chk2[7]; /* buffer for storing ascii representation of chksum2 */
char      exchk1[7]; /* buffer for storing a number of expected checksum 1 */
char      exchk2[7]; /* buffer for storing a number of expected checksum 2 */
char      errloc[7]; /* buffer for storing ascii rep of RAM error location */
char      errcnt[7]; /* buffer for storing ascii rep of RAM error count */
int      oldalgs; /* flag for storing current status of alarm status box */
int      almlfs; /* flag indicating new alarm status */
int      lars; /* flag set in calculations: if larsset detected */

/* Error flags for DMA test */

int      erMCR, erPAR, erDEF, erRSF, erFCT, erINT, erTEN;

int      error; /* 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 set in RAM */

char      *strinit[] = {0}; /* initialize pointer to buf */

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

int      nri[] = {055202, 047143, 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};

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

int      cmdn[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 6, 7, 8, 0};

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

char      *mdf[] = {"D", "N", "F", "d", "n", "l", "C2", "C1", "L", "P", "T", "c", "I", 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[] = "0000";

```

```
char      f9[ ]          "leas 08 *B%";
char      f10[ ]         "leas f10 /B%";
char      strn[ ]        "leas 08 *B%";
char      chr[ ]         "leas 24 *B%";
char      bel[ ]         "(07,0);
char      crlf[ ]        "\r\n";
char      blk[ ]         "010;040;010;0";
char      dlocf[ ]       "swor h:0014 16. 0100010010010";
char      nloc[ ]        "swor h:0015 16. 0100100010010";
char      eoloc[ ]       "swor h:0018 16. 0100010010010";
char      dloc[ ]        "swor h:00110 15. 01001001010010";
char      nloc[ ]        "swor h:00112 16. 0100010010";
char      lloc[ ]        "swor h:00114 16. 010010010";
char      c2log[ ]       "swor h:00116 16. 010010010010";
char      clloc[ ]       "swor h:00118 16. 010010010010";
char      ddist[ ]       "Fris 17 Possible values for D are 16,32. D=1004";
char      nlist[ ]       "Fris 17 Possible value for N are 2:N=4:n=N^1004";
char      relist[ ]      "Fris 17 Possible value for P are 10:P=256. P=1004";
char      dist[ ]        "Fris 17 Possible value for d are 0,0,0,10. d=1004";
char      nlist[ ]       "Fris 17 Possible value for n are 6,9,12,15,n=1. n=1004";
char      tlist[ ]       "Fris 17 Possible value for t are 3-15,t=5. t=1004";
char      c2list[ ]      "Fris 17 Possible value for C2 are 512,C2=204. C2=1004";
char      clist[ ]       "Fris 17 Possible value for C1 are 1,5,c1=1. c1=1004";
char      ddist[ ]       "swor h:lerna m:0011 26,IRCCD INTRUSION DETECTOR ALARM";
char      ddist[ ]       "0012;015;0117";
char      ddist[ ]       "0012;015 SYMBOL = VALUE + MEANING(0012 79;\012\015)";
char      ddist[ ]       "C1 D 00014 21.0 Number of cells monitored for stare time update \0014 79;\012\015";
char      ddist[ ]       "C1 L 00014 21.0 Fris 57;\012\015";
char      ddist[ ]       "C1 N 00016 21.0 Number of samples used in background time average \0016 79;\012\015";
char      ddist[ ]       "C1 F 00018 21.0 Separation of samples used in background time average. \012\015";
char      ddist[ ]       "C1 d 00010 21.0 Detection threshold scale factor \0010 79;\012\015";
char      ddist[ ]       "C1 n 00012 21.0 3 * memory interval length used in alarm decision \0012 79;\012\015";
char      ddist[ ]       "C1 L 00014 21.0 Number of space-time threshold violations to cause alarm\012\015";
char      ddist[ ]       "C1 U 00016 21.0 Upper threshold used to initiate stare time decrease \012\015";
char      ddist[ ]       "C1 L 00018 21.0 Lower threshold used to initiate stare time increase \012\015";
char      ddist[ ]       "C1=0111+0111+0111";
char      disp[ ]        "Fris 17 To change a parameter type symbol shown above \004";
char      seelst[ ]      "Fris 17 To see a list of other features, type L ";
char      blink[ ]       "Common h:lerna m:";
char      daterr[ ]      "Fris 17 Invalid input command, parameters, or sequence ";
char      date[ ]        "Fris 17 Invalid feature selection ";
char      sk2it[ ]       "012;015;012;015;0;
char      disp[ ]       "swor h:lerna m:0011 26,IRCCD INTRUSION DETECTOR ALARM";
char      disp3b[ ]      "00013 31 Additional Features ";
char      work[ ]        "swor 28;";
char      learn[ ]       "leas st./200000%;";
char      line9[ ]        "Fris 17 Please type number of desired feature \004";
char      chie[ ]        "0012\015Fris 17;";
char      linela[ ]      "C1) Disable audible alarm ";
char      linetb[ ]      "C1) Enable audible alarm ";
char      line2[ ]       "C2) Display output of interface A/D converter ";
char      line3[ ]       "C3) Programmable digital shift circuitry test ";
char      line4[ ]       "C4) DMU channel test ";
char      line5[ ]       "C5) RAM integrity test ";
char      line6[ ]       "C6) PROM checksum test ";
char      line7[ ]       "C7) Serial interface/terminal test ";
char      line8[ ]       "C8) Do nothing - return to parameter display ";
char      home[ ]        "Common h:lerna m:\015\down 4;Fris 28;";
char      help[ ]        "swor h:lerna m:28;NOTE Calculations are falling behind.";
char      binfor[ ]      "swor h:000110 3;";
```

```

char    out1oct[]    {"00110 c4:"};
char    ad1oct[]    {"0015 28:Output of A/D converter 0018 6:Binary\0018 67:Decimal:"};
char    fin1oct[]    {"00110 28:"};
char    rdsdis1[]    {"0015 16:Output of Programmable Digital Shift circuitry ":"};
char    rdsdis2[]    {"0018 6:Binary\0018 67:F1 F2\0018 67:Decimal:"};
char    test1[]      {"Title:0\05:"};
char    m1wer[]      {"tera m1tera w1wer 38 h:"};
char    m1mon1[]     {"tera m1tera w1mon 34 h h:"};
char    rmdis1[]     {"0015 22:PRGM Checksums ":"};
char    rmdis2[]     {"0018 9:Lower 4K\0018 62:Upper 4K:"};
char    rmdis3[]     {"00110 4:Actual 1 Expected\00110 57:Actual 1 Expected\00111 13:\00111 66:1:"};
char    a11oct[]     {"000111 4:"};
char    a121oct[]    {"000111 57:"};
char    a11oct[]     {"000111 18:"};
char    a121oct[]    {"000111 71:"};
char    rromok[]     {"00120 22:No checksum errors detected in PRGM ":"};
char    rkerin1[]    {"00120 15:Checksum error detected in lower 4K of PRGM ":"};
char    rkerin2[]    {"00122 18:Checksum error detected in upper 4K of PRGM ":"};
char    ramer[]      {"00110 20>Error detected in RAM at location ":"};
char    r1tehl[]     {"00112 6 005Proceed with test, anything else to terminate test:"};
char    ramer1[]     {"00120 15:Total number of errors detected in RAM was ":"};
char    ram1[]       {"0015 31:RAM Integrity Test:"};
char    roram[]      {"00120 18:No errors detected during RAM integrity test ":"};
char    h1tr1oct[]   {"000124 25 005Terminate test ":"};
char    h1tr1oct[]   {"00120 3 1r1s 23 005Proceed with test ":"};
char    r1jorn[]     {"00120 11 1:"};
char    b01[]        {"00120 30 30:"};
char    b02[]        {"00123 30 30:"};
char    b03[]        {"00124 30 30:"};
char    b04[]        {"00125 30 30:"};
char    b05[]        {"00126 30 30:"};
char    star1[]      {"*****:"};
char    star1[]      {"* * * * *:"};
char    targ1[]     {"00120 24 22: TARGET DETECTED 'mon h':"};
char    ntar1[]      {"00120 24 32:NO TARGET DETECTED'mon h':"};
char    alds1[]      {"00120 24 32: ALARM DISABLED 'mon h':"};
char    DMA1[]       {"0015 36:DMA Test:"};
char    m1wer[]      {"0018 19 a)\003R/W of MCR ":"};
char    m1wer[]      {"0018 19 a)\003R/W of MCR ":"};
char    BARer[]      {"00119 19 b)\003R/W of BAR ":"};
char    m1BARer[]    {"00119 19 b)\003R/W of BAR ":"};
char    DBRer[]      {"00119 19 c)\003R/W of DBR ":"};
char    m1DBRer[]    {"00119 19 c)\003R/W of DBR ":"};
char    RSTer[]      {"00111 19 d) Improper response to INIT signal detected ":"};
char    m1RSTer[]    {"00111 19 d) Proper response received from INIT signal ":"};
char    F1Ter[]      {"00112 19 e)\003function-status lines ":"};
char    m1F1Ter[]    {"00112 19 e)\003function-status lines ":"};
char    INTer[]      {"00113 19 f) End of transfer interrupt not detected ":"};
char    m1INTer[]    {"00113 19 f) End of transfer interrupt detected ":"};
char    T1Ter[]      {"00114 19 g)\003data transfer test ":"};
char    m1T1Ter[]    {"00114 19 g)\003data transfer test ":"};
char    atnd1[]      {"0015 25:A/D Converter Test Description ":"};
char    atnd2[]      {"0017 10 1) Twelve bit A/D converter output will be displayed on:"};
char    atnd3[]      {"0015 12:screen in both binary and decimal ":"};
char    ed1oct[]      {"0014 12:Programmable Digital Shift Circuitry Test Description ":"};
char    ed2oct[]      {"0016 16 1) Output of programmable digital shift circuitry will be displayed:"};
char    ed3oct[]      {"0017 13 on screen in both binary and decimal ":"};
char    ed4oct[]      {"0019 10 2) Function 1 and 2 lines will ster through all possible:"};
char    ed5oct[]      {"00110 1: combinations ":"};
char    ed6oct[]      {"00112 10 3) Number of bits displayed are related as follows ":"};

```


100

00502

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 troubleshooting the detector should the
** need ever arise.
**
*****
*/

```

```

/* include variable definitions file for use in compiling */
#include "def-irccd.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
   Tektronix 4024 terminal under control of this routine.
*/

```

```

main()
/* go to initialization routine */
initial()
start
/* test for incoming chara for rt serial interface */
if (testrtol == 0) goto write
/* get character from buffer and take appropriate action */
if (inrt)
    goto start
write
/* test if there is anything to write out to the tt. */
if (testrtol == 0) goto start
/* send characters to tt. */
sendrt
goto start

```

```

/* INVT - 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 necessary.
   Additional routines are then called which take the
   appropriate action to service these commands.
*/

```

```

invt()
char c;
int i;

```

```

01184      /* set pointer to serial interface input buffer */
c = 01177; /* remove any parity bits from incoming character */
if(c == 01001) /* delete line (i.e. ctrl-x, or del line keys) */
    puttlr(home); /* position cursor */
    puttlr(blank); /* clear line on screen */
    ttrnt=0; /* reset buffer count to 0 */
    ttrtrib=ttrib; /* reset pointer to beginning of input buffer */
    return;

01185      /* delete character (i.e. ctrl-h, or del char keys) */
    if(ttrnt) /* make sure not at start of line */
        ttrnt--; /* decrement input buffer counter */
        puttlr(bksp); /* put backspace on output stack */
        --ttrtrib; /* move pointer to input buffer back one char */
    }
    return;

01186      /* check if char is not CR (eol) */
    if(ttrnt==0) return; /* check if input buffer is full */
    *ttrtrib++ = c; /* put char in buffer and increment pointer */
    ttrnt++; /* increment input buffer counter */
    holdchar[c]=c; /* put char in temp location to allow placement on stack */
    holdchar[0]=0; /* add on eol char */
    puttlr(holdchar); /* put char on output stack */
    return;

01187      /* put last char of line in input buffer */
    ttrnt=0; /* set input buffer count back to zero */
    puttlr(crlf); /* put CR on output stack */
    send(); /* empty output buffers before executing command */
    /* parse and decode command line */
    parse();
    command=decoder(); /* command contains decoded command number */
    if(command==0 || c.decmd==0) /* invalid command, parameter, or combination */
        bds(); /* put error message on output stack */
        puttlr(disp2); /* put refresh message on screen */
        puttlr(home); /* reposition cursor */

01188      if(command==0 || c.decmd==0) /* check for invalid feature selection */
        puttlr(blank); /* clear bottom of screen */
        puttlr(dater2); /* put up error message */
        puttlr(s2l); /* skip 2 lines */
        puttlr(line2); /* refresh which feature line */

/* now branch to chosen command execution routine */
switch(command) {

    case 1: ddlist(); /* list possible values of D parameter */
        break;
    case 2: ndlist(); /* list possible values of N parameter */
        break;
    case 3: pllist(); /* 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: cllist(); /* 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(oldcmd) {
        case 1: dcmv(); /* go to change D parameter */
            break;
        case 2: ncmv(); /* go to change N parameter */
            break;
        case 3: pcmv(); /* go to change P parameter */
            break;
        case 4: dcmv(); /* go to change d parameter */
            break;
        case 5: ncmv(); /* go to change n parameter */
            break;
        case 6: pcmv(); /* go to change p parameter */
            break;
        case 7: c2mv(); /* go to change C2 parameter */
            break;
        case 8: c1mv(); /* go to change C1 parameter */
            break;
        case 9: lsmv(); /* go to routine which services additional features list */
            break;
        case 10: bds(); /* put up error message */
            puttty(disp2);
            puttty(home);
            break;
    }
    break;
    case 11: if(oldcmd != 10) { /* check if line only consisted of CR */
        /* put up additional features list */
        addfeat();
    }
    else {
        if (oldcmd == 9) { /* put up initial parameter display */
            outdsp1;
        }
        else {
            bds(); /* put up error message */
            puttty(disp2);
            puttty(home);
            oldcmd=1;
        }
    }
    break;
}
/* reset pointers to */
tt1ptr=tt1buf;
return;
}

/* PARSE - routine for parsing command line.
   Place command line into the array "strng"
   as
   parter

```

```

int i;
tltprob=tlbub; /* set pointer to beginning of buffer */
i=0; while(tltprob<015) { /* check for CR */
    *strings[i]=tltprob; /* put char into array */
    *strings[i+1]=0; /* add eol char */
    return;
}
for(i=tltprob==040; tltprob++;) /* skip spaces */
if(tltprob==015) return; /* check for CR */
for(i=0; tltprob!=040 && (tltprob!=015); i++) {
    if(i>9) i=9; /* i points to each element in array */
    *strings[i]=tltprob++; /* put chars into array */
}
*strings[i+1]=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(*strings[i]==*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(*strings[i+1]==*cmds[7]+1) k=5; /* check if C2 */
        else if(*strings[i+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(*strings[0]>=060 && *strings[0]<=071 || *strings[0]==056) k=10;
    }
    /* decode numeric string in appropriate manner */
    if(k==10) {
        m=ascfoni(); /* "ascfoni" returns mantissa in "datares" */
        if(m==1) k=0; /* m indicates whether or not there was an error in conversion */
    }
    if(aldcmds==9 && k!=10) k=0; /* set error if not numeric when expecting */
    if(*strings[i]==015) k=11; /* check if command is CR only (to toggle display, etc.) */
    return(k);
}

/* ASCFON - routine for converting numeric ascii strings into
appropriately formatted binary numbers
*/
ascfoni()

```

```

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

```

```

power=1000;
for(i=0; i<8; i++) {
    frac=frac*10;
    if(frac>=power) { /* check which bits to set in datares */
        datares=(datares<<i) | 01; /* set these bits */
        frac-= power;
    }
    else datares=datares<<1; /* don't set bit in datares */
}
datares++; /* round off fractional bits */
datares=datares>>1;
/* datares is in binary format dddddd 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" "bufin" and "bufout"
   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 cnt=0; /* bytecnt, *stackp;
    char c; /* *bufin, *bufout;
    d=TI0BUF;
    tlostat=TI0STAT;
    for(i=0; i<10; i++) {
        if(i==0) { /* determine which stack is being sent out */
            *bufout=&outst2; /* set output pointer to interrupt stack */
            *bufin=&in2; /* set input pointer to interrupt stack */
            bytecnt=btent2; /* bytecnt tells which byte in the current ascii string is being sent to tty */
            *stackp=&outst2; /* use interrupt output queue */
        }
        else {
            *bufout=&outst1; /* set output pointer to user stack */
            *bufin=&in1; /* set input pointer to user stack location */
            bytecnt=btent1; /* set which byte in string to start with */
            *stackp=&outst1; /* use user output queue */
        }
        cnt=0;
        while(*bufin != *bufout) { /* test if stack empty */
            while(*tlostat==0) { /* test status of ready bit on serial interface */
                c=**bufout+bytecnt; /* get next character */
                if(i==0) { /* service interrupt stack */
                    if(c==33) cnt=1; /* check for command character(!)... */
                    else if(c==59 && cnt!=0) {
                        cnt=0;
                        *c; /* send out ! character */
                    }
                }
                /* send out some nulls to let term catch up */
                for(j=0; j<10; j++) {
                    while(*tlostat==0);
                    *c=0;
                }
                while(*tlostat==0);
            }
            *bufout=c;
            cnt++;
            *bufin++;
            *bufout++;
        }
    }
}

```

```

    }
}
else {
    if(c==01) { /* send out "rum" sequence... */
        for(i=0; i<5; i++) {
            *d=rnm(i);
            while(*ttostat==0);
        }
        c=0;
    }
    if(c==02) { /* send "No error detected in" */
        for(i=0; i<22; i++) {
            *d=Noen(i);
            while(*ttostat==0);
        }
        c=0;
    }
    if(c==03) { /* send "Error detected in" */
        for(i=0; i<19; i++)
            *d=Err(i);
        while(*ttostat==0);
        c=0;
    }
    if(c==04) { /* send "home" sequence */
        for(i=0; i<20; i++) {
            *d=home(i);
            while(*ttostat==0);
        }
        c=0; /* send null char */
    }
    if(c==05) { /* send "Hit RETURN" sequence */
        for(i=0; i<10; i++) {
            *d=hitrt(i);
            while(*ttostat==0);
        }
        c=0;
    }
    if(c==0137) { /* check for underscore repeat */
        i=*((pout+1)+b_tecnt); /* check how man. */
        for(i=0; i<i; i++)
            *d=0137;
        while(*ttostat==0);
        c=0;
    }
    *d=0;
    b_tecnt++; /* go to next char in string */
    c=*((pout+b_tecnt));
    if(c==0) { /* check for end of line char */
        b_tecnt=0;
        pout++;
        if(pout>stacket+40) pout=stacket; /* wrap around circular queue */
    }
    if(i==0) { /* reset pointers */
        b_tecnt2=pout;
        b_tecnt2=b_tecnt;
    }
}

```

```

    else {
        b=bt1=ptout;
        bcnt1=b-1;
    }
    return;
}

```

```

/* PUTTY1 - routine to place location of outgoing ascii on
   output queue list no. 1
   This is information from user controlled functions
*/

```

```

*
outtk1(100);
int loc;
*bt1=100; /* set pointer to start of string */
bct1++;
*bt1=(outtk1+40); bct1=outtk1; /* check for wrap around on queue */
}

```

```

/* PUTTY2 - routine for placing location of outgoing ascii from
   interrupt service routine on output queue list no. 2
*/

```

```

*
outtk2(100);
int loc;
*bt2=100; /* set pointer to start of string */
bct2++;
*bt2=(outtk2+40); bct2=outtk2; /* check for wrap around on queue */
}

```

```

* INITIAL - initialization routine for IPCCD intrusion
   detector. Called at turn on time and when routine is
   restarted after performing diagnostic tests
*/

```

```

*
initial:
int *d;
e term int dummy;
/* disable interrupts temporarily */
d=CSR;
*d=0177667;
d=INTVEC;
/* place starting address of dummy routine in DMA interrupt
   vector location */
*d+= &dummy;
*d=0;
d=ICR;
*d=0177400; /* set DMA WCR to initial value --250*/
d=BAR;
*d=rbufl; /* set DMA BAR pointing to rbufl for first packet*/
stinstat=TT*STAT; /* set pointer to serial int. input status bytes*/
oldcmd=1;
aldrisab=0;
fla=0;
*trbit=0; /* set pointer to input buffer */

```



```

datares=1;
dat:trns(0)=061;
dat:trns(1)=062;
dat:trns(2)=0;
ppsr(0);
/* n value = 12 */
nsr(0);
/* C2 value = 480 */
datares=480;
dat:trns(0)=064;
dat:trns(1)=070;
dat:trns(2)=060;
dat:trns(3)=0;
c2psr(0);
cond(0); /* write out output bytes */
/* C1 value = 280 */
datares=280;
dat:trns(0)=062;
dat:trns(1)=063;
dat:trns(2)=0;
dat:trns(3)=0;
c1psr(0);
/* t value = 8 */
datares=8;
dat:trns(0)=070;
dat:trns(1)=0;
trns(0);
/* t value = 17 */
/* format of d is nnnn nnnnnn */
datares=141;
dat:trns(0)=066;
dat:trns(1)=056;
dat:trns(2)=061;
dat:trns(3)=0;
dat:trns(5)=0;
psr(0);
cond(0);
trns(0);
/* enable interrupts from DMG */
/* hit so hit */
/* will wait for a few microseconds to make sure interrupt has hit */
/* set alarm disable count */
aldrab=brabrap+1; /* this should do it */
*hl 01
/* raise function three line */
*dl 010;
return;
)

/* DDLIST - routine to list "D" values */
ddlist(0;
psr(0);
outlv1:ddlist;
olddcmd=1;
)

/* NDLIST - routine to list "N" values */
ndlist(0;

```

```

      print
      putv(1,0,1,0);
      oldcmd=2;
    }

    /* PLIST - routine to list "P" values */
    plist() {
      bds();
      putv(1,0,1,0);
      oldcmd=3;
    }

    /* QLIST - routine to list "Q" values */
    qlist() {
      bds();
      putv(1,0,1,1);
      oldcmd=4;
    }

    /* MLIST - routine to list "M" values */
    mlist() {
      bds();
      putv(1,0,1,0);
      oldcmd=5;
    }

    /* ULIST - routine to list "U" values */
    ulist() {
      bds();
      putv(1,0,1,1);
      oldcmd=6;
    }

    /* ZLIST - routine to list "Z" values */
    zlist() {
      bds();
      putv(1,0,2,1);
      oldcmd=7;
    }

    /* ALLIST - routine to list "A" values */
    alist() {
      bds();
      putv(1,0,1,1);
      oldcmd=8;
    }

    /* DGET - routine to change "D" parameter */
    dget() {
      int i;
      /* check if allowed value of D parameter */
      if (datares[i] != %d) {
        bds(); /* put up error message */
        putv(1,0,1,1);
        return;
      }
      /* temporarily disable calculation routines */
      intflg=1;
      bsd=datares[i]; /* set new b value */
      /* put ascii string of parameter value in buffer */

```

```

for(i=0; datstrns[i].i++;) b1datstr[i]=datstrns[i].
b1datstr[i]=0.
n1b1spara=256*b1sd. /* set number of terms used in computing background spat avg */
b1sd=b1sd/1. /* 240 value used in calculation routine */
/* set number of shifts to divide by D */
if(b1sd=16) dshft=-4.
else dshft=-5.
/* enable calculations */
in1f1s=0.
out1f1f1ddisc1
out1f1f1b1s1tr1. /* send new value of D to terminal */
/* refresh bottom of screen */
bsc1.
out1f1f1disc2.
i1dcomnd=10.
return
}

/* PPSR1 - routine to change "N" parameter */
ppsr1() {
in1f1.
/* check for valid N parameter values */
if(datanes[2] || datanes[40]) {
b1s1. /* put up error message on screen */
out1f1f1e1s1.
return.
}

/* temporary calculation routine */
in1f1s=1.
b1s=datanes /* set new value of N */
/* look up par1(N+1) in table */
parane1[b1s-1]
thresh1 /* compute new threshold table used in computations */
for(i=0; datstrns[i].i++;) b1datstr[i]=datstrns[i].
b1datstr[i]=0.
u1s=512*b1s. /* value to tell when vector holding past data is full */
p1nt=0. /* counter to tell which nth of computations on */
r1d1sh=b1s+10001. /* value used to aid in rounding off operation */
v1ptr= &x1. /* set pointer to start of x vector */
m1ptr= &m1. /* set pointer to start of m vector */
for(i=0; i<(b1s/32).i++;) x[i]=0. /* clear x vector */
for(i=0; i<(1024).i++) m[i]=0. /* clear m vector */
v1ntr=0. /* clear pointer to element in vbuf */
n1=b1s*b1s. /* compute NxF */
a1disab=n1. /* set alarm disable counter */
/* enable computations */
in1f1s=0.
/* send new value to tty */
out1f1f1n1s1.
out1f1f1b1s1tr1.
/* refresh bottom of screen */
bsc1.
out1f1f1disc2.
i1dcomnd=10.
return
}

/* PPSR2 - routine to change "P" parameter */
ppsr2() {
in1f1.

```

```

/* check if valid P parameter value */
if(datares<10 || datares>256) {
    bds(); /* send error message to tty */
    puttty("\n");
    return;
}

/* temporarily disable computations */
initfs=1;
bias=datares; /* set new value of P */
for(i=0; datstrng[i] != '\0'; i++) biasstr[i]=datstrng[i];
biasstr[i]=0;
/* compute how computational load should be divided */
/* pnorm tells how many computations will be done the first
   nl-1 iterations while plast tells how many will be done the
   nth time nl may or may not be equal to P depending on the
   current value of P.
*/
nl=bias-1;
pcnt=0;
pnorm=(512/bias)+1000;
tpnorm=pnorm*0.1;
plast=256-pnorm*nl;
if(plast<25) { /* allow maximum of 25 computations each time */
    pnorm++;
    tpnorm=pnorm*0.1;
    plast=256-pnorm*nl;
}
while(plast!=0) {
    nl--;
    plast=256-pnorm*nl;
}
nl++;
xptr=0; /* set pointer to x vector */
for(i=0; i<(128/8); i++) x[i]=0; /* clear x vector */
xptr=16; /* set pointer to start of a vector */
/* clear a vector */
for(i=0; i<(24/4); i++) m[i]=0;
/* set pointer to starting location in bus */
iptr=0;
nexttransit; /* compute N+P */
alldatans; /* set alarm routine to N+P */
/* enable calculations */
initf=0;
/* send new P value to tty */
puttty("\n");
puttty("\n");
hprt(); /* refresh bottom of screen */
puttty("\n");
oldcount=0;
return;
}

/* BSR - routine to change "d" parameter. */
dpsm();
int i;
/* check for valid d parameter values */
if(datares<10 || datares>02400) {
    bds(); /* send error message to tty */
    puttty("\n");
    return;
}

```

```

smn=dalares; /* set new value of d parameter */
for(i=0; datstrns[i].i++;) smdstr[i]=datstrns[i].
smdstr[i]=0;
thresh(i); /* compute new threshold table for use in computations*/
/* send new d value to tty */
puttty(dloc);
puttty(smdstr);
bcs();
putty(disp2);
oldsmn=smn;
return;
}

/* NSRV - routine to change "n" parameter */
nrv();
{
  int i;
  /* check for valid n parameter values */
  if(dalares!=6 && dalares!=9 && dalares!=12 && dalares!=15) || dalares<smn) {
    bds(); /* put error message on tty */
    puttty(nlist);
    return;
  }
  /* temporarily disable computations */
  intls=1;
  smn=dalares; /* set new n value */
  for(i=0; datstrns[i].i++;) smnstr[i]=datstrns[i].
  smnstr[i]=0;
  mask=(0177700)/((5-smn/3)) + 01 /* set up mask for some routines*/
  /* aldisab(smn/3) aldisab(smn/3); /* set alarm disable */
  /* enable computations */
  intls=0;
  /* send new n value to tty */
  puttty(nloc);
  puttty(smnstr);
  bcs();
  putty(disp2);
  oldsmn=smn;
  return;
}

/* TSRV - routine to change "t" parameter */
tsrv();
{
  int i;
  /* check for valid t parameter values */
  if(dalares<smn || dalares<3 || dalares>15) {
    bds(); /* send error message to tty */
    puttty(tlist);
    return;
  }
  /* set new t value */
  smt=dalares;
  for(i=0; datstrns[i].i++;) smtstr[i]=datstrns[i].
  smtstr[i]=0;
  /* send new t value to tty */
  puttty(tloc);
  puttty(smtstr);
  bcs();
  putty(disp2);
  oldsmn=smn;
}

```

```

return
}

/* DISRV - routine for changing "C2" parameter. */
disrv() {
    int i;
    /* check for valid C2 parameter values */
    if (datares[0] < 1 || datares[0] > 511) {
        bds(); /* send error message to tty */
        puttty(c2lst);
        return;
    }
    /* set new C2 parameter value */
    c2=datares;
    for(i=0; datstrns[i] != '\0'; i++) c2str[i]=datstrns[i];
    c2str[i]=0;
    /* enable/disable computation */
    intflag=1;
    /* compute multiplier of C2 for use in state time changes */
    cc1[3]=2;
    for(i=2; i>0; i--) cc2[i]=cc2[i+1]*C1;
    /* enable computations */
    intflag=0;
    /* send new C2 value to tty */
    puttty(c2loc);
    puttty(c2str);
    bds();
    puttty(disp2);
    oldcmd=10;
    return;
}

/* DISRV - routine for changing "C1" parameter. */
disrv() {
    int i;
    /* check if valid C1 parameter value */
    if (datares[0] < 1 || datares[0] > 511) {
        bds(); /* send error message to tty */
        puttty(c1lst);
        return;
    }
    /* set new value of C1 parameter */
    c1=datares;
    for(i=0; datstrns[i] != '\0'; i++) c1str[i]=datstrns[i];
    c1str[i]=0;
    /* enable/disable computation */
    intflag=1;
    /* compute multiplier of C1 for use in state time changes */
    cc1[3]=1;
    for(i=2; i>0; i--) cc1[i]=cc1[i+1]*C1;
    /* enable computations */
    intflag=0;
    /* send new C1 value to tty */
    puttty(c1loc);
    puttty(c1str);
    bds();
    puttty(disp2);
    oldcmd=10;
    return;
}

```

```

/* LSRV - routine to do "other features" */
lsrv() {
  int c,d;
  d=TTIBUF;
  switch(datar) {
    /* change status of audible alarm */
    case 1: audalrm=audalrm+100;
             break;
    /* go to routine which tests the A to D converter */
    case 2: adtst();
             return;
    /* go to routine which tests the programmable digital shift logic */
    case 3: pdtst();
             return;
    /* go to routine which displays the DMA test instructions */
    case 4: dmaint(); /* put up DMA test instructions */
             while(*testinstalC==0) send(); /* wait for halt or skip test */
             c=ed & 0177;
             if(c != 015) addfeat(); /* put up additional test lists */
             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: termstst();
             return;
  }
  /* go back to the original display */
  outd=ol;
}

/* ADDEFeat - 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() {
  puttty("disc3");
  puttty("disc3b");
  puttty("orle");
  puttty("skip");
  /* decide whether to put up audible alarm enable or disable */
  if(audalrm==0) puttty("line1a");
  else puttty("line1b");
  puttty("skip");
  puttty("line2");
  puttty("skip");
  puttty("line3");
  puttty("skip");
  puttty("line4");
  puttty("skip");
  puttty("line5");
  puttty("skip");
  puttty("line6");
  puttty("skip");
}

```

```

puttlv('line7');
puttlv('skip');
puttlv('line8');
puttlv('blank');
puttlv('sk21');
puttlv('line9');
/* draw box for target messages */
alarm() /* fill up box with appropriate message */
oldconds=0;
)

```

```

/* PUTDSP1 - routine for putting initial parameter display list on the
screen. This is the display that is first seen at turn on time
*/

```

```

display():
drawtbl() /* draw table at top of screen */
send(); /* empty output queues */
/* fill in various parameter values */
puttlv('dd1oc');
puttlv('bigdstr');
puttlv('nnloc');
puttlv('bignstr');
puttlv('exploc');
puttlv('bigestr');
puttlv('dloc');
puttlv('smdstr');
puttlv('nloc');
puttlv('smnstr');
puttlv('lloc');
puttlv('smtstr');
puttlv('c2loc');
puttlv('c2str');
puttlv('elloc');
puttlv('elstr');
/* fill in bottom of screen */
btop();
puttlv('disp2');
box(); /* draw box for target messages */
alarm(); /* fill up box with appropriate message */
oldconds=10;

```

```

/* DISP.A1 - 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
*/

```

```

display():
puttlv('dd11');
puttlv('dd12');
puttlv('dd13');
puttlv('dd14');
puttlv('dd15');
puttlv('dd16');
puttlv('dd15');
puttlv('dd18');

```


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; divide shift
/* set up DMA to load in 1 word from interface ... */
d=WCR;
*d=0177777;
d=BAR;
*d= *atodbyt;
d=CSR;
*d= 01 /* bit 150 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 */
atoder(atodbyt-atdoct);
/* display binary and decimal versions of word on screen */
puttyr(binloc);
puttyr(atdbin);
puttyr(10ctloc);
puttyr(atdoct);
/* clear output buffers */
b=0;
}

/* AFODINS - 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() {
puttyr(10dd1);
puttyr(10atod1);
puttyr(10atod2);
puttyr(10atod3);
puttyr(10bit1);
b=0; /* draw box for target messages */
alarm(); /* fill box with appropriate message */
send(); /* empty output queue */
}

/* PDSTST - routine to test the programmable 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.

```

/*
pdslist() {
int *b, fln=1;
char c, *d;
d=TTIBUF;
pdsins(); /* put up pds instructions ... */
/* wait for continue symbol (return) or skip test (anything else) */
while(*stinsal!=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 interrupts */
b=CSR;
*b= 0177677;
/* set function lines initially to true */
fln=06;
*b=1 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_alod(); /* 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;
eb= fln;
}

if(stinsal==0) goto testit; /* look for break signal */
c= *d
/* terminate test and reset program */
putttyl(c) /* A to D initialization routine */
}

```

/* PDSINS - routine to display PDS test instructions
This routine lists a brief description of the test of

The Programmable Digital Shift Logic circuitry.

```

00000000
00000001
00000010
00000011
00000100
00000101
00000110
00000111
00001000
00001001
00001010
00001011
00001100
00001101
00001110
00001111
00010000
00010001
00010010
00010011
00010100
00010101
00010110
00010111
00011000
00011001
00011010
00011011
00011100
00011101
00011110
00011111
00100000
00100001
00100010
00100011
00100100
00100101
00100110
00100111
00101000
00101001
00101010
00101011
00101100
00101101
00101110
00101111
00110000
00110001
00110010
00110011
00110100
00110101
00110110
00110111
00111000
00111001
00111010
00111011
00111100
00111101
00111110
00111111
01000000
01000001
01000010
01000011
01000100
01000101
01000110
01000111
01001000
01001001
01001010
01001011
01001100
01001101
01001110
01001111
01010000
01010001
01010010
01010011
01010100
01010101
01010110
01010111
01011000
01011001
01011010
01011011
01011100
01011101
01011110
01011111
01100000
01100001
01100010
01100011
01100100
01100101
01100110
01100111
01101000
01101001
01101010
01101011
01101100
01101101
01101110
01101111
01110000
01110001
01110010
01110011
01110100
01110101
01110110
01110111
01111000
01111001
01111010
01111011
01111100
01111101
01111110
01111111
10000000
10000001
10000010
10000011
10000100
10000101
10000110
10000111
10001000
10001001
10001010
10001011
10001100
10001101
10001110
10001111
10010000
10010001
10010010
10010011
10010100
10010101
10010110
10010111
10011000
10011001
10011010
10011011
10011100
10011101
10011110
10011111
10100000
10100001
10100010
10100011
10100100
10100101
10100110
10100111
10101000
10101001
10101010
10101011
10101100
10101101
10101110
10101111
10110000
10110001
10110010
10110011
10110100
10110101
10110110
10110111
10111000
10111001
10111010
10111011
10111100
10111101
10111110
10111111
11000000
11000001
11000010
11000011
11000100
11000101
11000110
11000111
11001000
11001001
11001010
11001011
11001100
11001101
11001110
11001111
11010000
11010001
11010010
11010011
11010100
11010101
11010110
11010111
11011000
11011001
11011010
11011011
11011100
11011101
11011110
11011111
11100000
11100001
11100010
11100011
11100100
11100101
11100110
11100111
11101000
11101001
11101010
11101011
11101100
11101101
11101110
11101111
11110000
11110001
11110010
11110011
11110100
11110101
11110110
11110111
11111000
11111001
11111010
11111011
11111100
11111101
11111110
11111111

```

* TERMIST - 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
 */
 termist() {
 int *b,i,k;
 char c,*d;
 i=TTIBUF;
 termist(); /* put up terminal test instructions */
 /* wait for continue symbol (return) or skip test (any other) */
 while(*b!=0) send(); /* refresh target messages while waiting */
 b = &0177;
 if(c != 015) { /* check if not CR */
 addreal(); /* put up additional features list */
 return;
 }
 /* perform actual int/term test: */
 /* disable interrupts */
 p=CSR;
 b=0177677;
 /* test terminal with standard 4024/4025 test */
 outttb(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<2000; i++);
 /* construct an E1 character line for repeated display on screen */
 b=052;
 for(i=0; i<81; i++) {
 if(i%4) = 0127; workin[i]=i;
 else workin[i]=050; /* this 4024 command character (1) */
 }
 outttb(1, 0); /* end of line char */
 outttb(1, b);
 send(); /* empty output queue */

```

/* wait for proceed (an. keyboard entry) */
initial: linstat:=0;
ce:=4d;
/* fill monitor screen with 81 character lines */
puttlr(mkmon); /* make screen all monitor space */
while(*stinstat:=0) ( /* check serial interface status */
    puttlr(runkin); /* 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 */
);
ce:=4d;
/* fill workspace with 81 character lines. */
puttlr(mkwork); /* make screen all workspace (except last line) */
while(*stinstat:=0) ( /* check serial interface status */
    puttlr(runkin); /* 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 */
);
ce:=4d;
/* reset routine and return */
initial(); /* go to initialization routine */
)

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

/* PROMISI - 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.
*/
promisi() (

```

```

int *b; cks1 = cks2; ckslen = cks2len;
char *d;
d = "TIME";
main: /* put up from test instructions */
/* wait for continue symbol(return) or end test(anything else) */
while(*cinstal!=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 from board
*/
/* disable interrupts */
b = SR;
b = 0177677;
/* label screen */
puttty(1, ddi1);
puttty(1, ermdis1);
puttty(1, ermdis2);
puttty(1, ermdis3);
send(); /* empty output queues */
/* compute actual checksums */
b = TL4; /* set pointer to address of start of lower 4k */
chksum1 = 0; /* clear checksum accumulator */
while(b != ENDLFX) { chksum1 = *b++; /* compute lower 4k checksum */
/* b is now left pointing at start of upper 4k */
chksum2 = 0; /* clear upper 4k checksum accumulator */
while(b != ENDUFX) { chksum2 = *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 */
e-cks1 = *b++;
e-cks2 = *b;
/* get from error flags */
ckslen = 0;
cks2len = 0;

/* compare actual with expected checksums */
if(chksum1 != e-cks1) ckslen = 1;
if(chksum2 != e-cks2) cks2len = 1;
/* display actual and expected checksums */
/* convert to ascii (octal) */
atocvt(chksum1, ck1);
atocvt(chksum2, ck2);
atocvt(e-cks1, eck1);
atocvt(e-cks2, eck2);
/* put strings in output queue */
puttty(1, ck1len);
puttty(1, ck1);
puttty(1, eck1len);
puttty(1, eck1);
puttty(1, ck2len);
puttty(1, ck2);
puttty(1, eck2len);
puttty(1, eck2);
/* display no error message if appropriate */
if((ckslen == 0) && (cks2len == 0)) puttty(1, promok);
/* display error messages if error has been detected */

```

```

else {
    if(csk1er != 0) puttyvl(csk1er1);
    if(csk2er != 0) puttyvl(csk1er2);
}

puttyvl(hits1);
puttyvl(home);
send(); /* empty output queues */
/* wait for break */
while(*tinstat==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
*/
promins() {
    puttyvl(dd1);
    puttyvl(prom1);
    puttyvl(prom2);
    puttyvl(prom3);
    puttyvl(prom4);
    puttyvl(hittst);
    box(); /* draw box for target messages */
    alarm(); /* fill box with appropriate message */
    send(); /* empty output queues */
}

/* RAMTEST - 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.
*/
ramtest() {
    int *b1,k,erfls,ercont,erloc;
    register int *r1,r2,r3; /* define 11-03 registers */
    char c,*d;
    d=TTBUF;
    clrscr(); /* display RAM test instructions */
    /* wait for continue/return or skip test (any other key) */
    while(*tinstat==0) send(); /* refresh target message while waiting */
    c = *d + 0177;
    r1 = 015; /* check if not CR */
    addfeat(); /* put up additional feature list */
    return;

    /* perform actual Ram test */
    /* disable interrupt */
    b=CSR;

```

```

404 0177477
/* put up initial display headings */
puttyv(0d11);
puttyv(ram1);
send(); /* empty output queues */
err1=erent=0; /* set error flag and error count to 0 */
r1= STRRAM; /* 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: err1=1;
    erent++; /* increment error count */
    erloc=r1; /* read error location back from register */
    /* convert error location to ascii (octal) */
    atodc(erloc,erlok);
    /* put error location string on output queue */
    puttyv(ramer);
    puttyv(erlok);
    puttyv(1rcoeh);
    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 ER */
    /* convert and display the error count (decimal) */
    atodc(erent,erent);
    /* put error count on output queue */
    puttyv(1rament);
    puttyv(erent);
    send(); /* empty output queues so error count is displayed */
    for(k=0; k<5; k++) for(i=0; i<30000; i++); /* delay so erent may be seen */
    initial(); /* go to initialization routine */
    return;
next: r1++; /* go to next word in RAM */
}
/* display end of test messages */
if(erent == 0) puttyv(1ram); /* no error message */
/* convert and display total error count (decimal) */
else {
    atodc(erent,erent);
    puttyv(1rament);
    puttyv(erent);
}
puttyv(1h1str);
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() {
    puttyv('ddll');
    puttyv('ramin1');
    puttyv('ramin2');
    puttyv('ramin3');
    puttyv('ramin4');
    puttyv('ramin5');
    puttyv('ramin6');
    puttyv('ramin7');
    bsc(); /* 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]=0; /* 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]='-'; /* 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) { /* so 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[i]=060; /* make sure at least one 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

```

/*
atoccl(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[1]=(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(alds1); /* put alarm disabled message in output queue */
break;
case 1 puttty2(targ1); /* put target detected message in output queue */
break;
case 2 puttty2(notar1); /* put no target detected message in output queue */
break;
)
)

```

/* IRQOUT - 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 audible flag is set (from selection of feature 1)

```

/*
irqout() (
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!=oldalrm) (
oldalrm=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() {
    putty2(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() {
    putty1(box1);
    putty1(star21);
    putty1(box2);
    putty1(star2);
    putty1(box3);
    putty1(star2);
    putty1(box4);
    putty1(star2);
    putty1(box5);
    putty1(star21);
}

/* DMA TEST - 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
*/
dma_test() {
    char c = 46;
    clrbuf();
    loop: dma_test(); /* go to routine which sets up for call to assembly code */
    while(*stinstr() != 0); /* wait for repeat character */
    c = *d;
    goto loop; /* go back and perform DMA test again */
}

```

/* DPATS - 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 MCR
- (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) Transferring 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 necessary for proper execution of several of the tests.

```

/*
dma1s() {
int *b;
/* perform actual DMA test. */
/* disable dma interrupts. */
b=CSR;
*b=k 0177677;
/* label screen */
puttvl(ddll);
puttvl(DMA11);
puttvl(hitrop);
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 */
erMCR=erBAR=erDBR=erRST=erFCT=0;
dma(); /* so to assembly language DMA test routine ... */
/* display results */
if(erMCR>0) puttvl(MCRe); /* error in R/W of MCR */
else puttvl(noMCR); /* no error in R/W of MCR */
if(erBAR>0) puttvl(BARe); /* error in R/W of BAR */
else puttvl(noBAR); /* no error in R/W of BAR */
if(erDBR>0) puttvl(DBRe); /* error in R/W of DBR */
else puttvl(noDBR); /* no error in R/W of DBR */
if(erRST>0) puttvl(RSTe); /* improper response to INIT */
else puttvl(noRST); /* proper response to INIT */
if(erFCT>0) puttvl(FCTe); /* error in function-status lines */
else puttvl(noFCT); /* no error in function-status lines */
if(erINT>0) puttvl(INTER); /* no interrupt detected */
else puttvl(noINT); /* no interrupt detected */
if(erTRND>0) puttvl(TRNRe); /* error in data transfer */
else puttvl(noTRN); /* no error in data transfer */
send(); /* empty output queues so error messages appear on screen */
}

```

/* DPAINS - 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.

```

/*
dmain1s() {
puttvl(ddll);

```

```

puttvl(dma1);
puttvl(dma2);
puttvl(dma3);
puttvl(dma4);
puttvl(dma5);
puttvl(dma6);
puttvl(dma7);
puttvl(dma8);
puttvl(hithit);
bn(); /* draw box for status messages */
alarm(); /* fill box with current status message */
puttvl(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() {
puttvl(blink);
puttvl(seelst);
puttvl(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() {
puttvl(blink);
puttvl(daterr);
puttvl(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) Interrupt 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 assembler's vocabulary

rti = 000002
nop = 000240
reset = 000005
ntes = 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

_start = + 20000

_start: mov $400,r0 /setup vector area with halts
1: clr -(r0)
mov r0, -(r0)
lsl 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),-1(sp)
lsl (r0)+
mov r0,2(sp)

```

```

    r0:    PC+main
    cmp    r0, r0, r0
    mov    r0, r0
    r0:    PC+main
    r0:    exit

```

```

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

```

```

    r0:    r0
    r0:    r0

```

```

    r0:    r0
    mov    r5, r0
    mov    r5, r5
    mov    r4, r0
    mov    r3, r0
    mov    r2, r0
    mov    r1, r0
    mov    r0, r0

```

```

    r0:    r0
    mov    r5, r1
    mov    r4, r1
    mov    r3, r1
    mov    r2, r1
    mov    r1, r5
    mov    r0, r5
    r0:    r0

```

```

/* C register save and restore

```

```

    r0:    r0
    r0:    r0

```

```

/* C routine to handle an interrupt occurring at end of data packet
* This routine is entered from the interrupt routine, the DMW
* heralding that a data transfer has been initiated. The DMW 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

```

```

    r0:    r0, r0
    mov    r1, r0
    r0:    r0

```

```

/* Check if falling behind

```

```

    r0:    r0
    mov    r0, r0

```

```

/* Print error message

```

```

    r0:    r0
    mov    r0, r0

```

/ Toggle function 3 line to stop start convert pulses

```

pic    $10,+$172414
nop    / wait a while before switching to line back
bis    $10,+$172414
mov    $177400,+$0172410    / load wdr register with -256

```

/ Set so bit for next load

```

bis    $101,+$172414

```

Determine which buffer to load next packet into

```

tst    +bufnum
bne    +bf2

```

Load next one into +buf2

```

set    $1, +bufnum
mov    $+buf2,+$0172412    / load BAR
mov    $+buf1,+$0172410    / set pointer to current buffer
br     cmpt

```

/ Load next packet into +buf1

```

set    $1, +bufnum
mov    $+buf1,+$0172412    / load BAR
mov    $+buf2,+$0172410    / set pointer to current buffer

```

/ Check if computations should be skipped due to parameter change

```

cmpt    tst    +intflg
        beq    crn
        inc    +aldisab    / disable alarm for at least one packet
        br     noern

```

/ Call computational routine crunch

```

crn     jsr    +crunch

```

/ Call C routine which updates status box on screen

```

noern   jsr    +cstatus
        clr    +flag    / clear flag used in determining if calculations flow

```

/ Restore r1 and r0

```

mov     +r0,r1
mov     +r1,r0
rti

```

DUMMY - 1 - connect 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 merely sets the interrupt vector pointing to the ENDIF
/ routine and thus prevents entering the computations before an

```

actual data packet is received

- dummy = 0 \$=endp \$=0124

Set initial values of function lines:

 bld = \$6.\$172414
 b1s = +trsf.\$172414
 rt1

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 initiation of 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 of

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 referred 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. *icptr* 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 precision 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 precision 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.

```

    label crunch
    text
    crunch
        isb    r5,csr / call register save routine
        mov    r5,+lwp2 / save r5

    / Compute background spatial average

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

        mov    +ntbbkspan,r0

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

        mov    +rptr,r1

    / lbad 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    +lbad,r4
        clr    r3
        clr    r2
loopa: add    (r1),r3 / add sample into accumulator
        adc    r2    / add carry to compute double precision 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 precision sum -dshift
    / tells how many places to shift for the current D value

        ashc    +dshift,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. (oldtreff) contains this
    / information

```

```

add    +oldtref,r0

/ Check for stare time decrease
mov     r3,r0
btl     stime

/ Decrease stare time if possible
inc     +fstat / set flag indicating stare time change

/ Test if stare time can be decreased any further
lsl     +tref
beq     date    / if not, do not change
sub     $2,+tref / change tref (not sent out yet)

/ Go to node which sends out new stare time via function lines 1 2
imr     update

/ Check for stare time increase
stinc   mov     $+c1,r0 / set r0 pointing to start of CI 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     +trefml,+oldtref
        mov     +tref,+trefml

/ 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', 'elast', and 'nl'. There are 'pnorm' calculations

```

```

/ performed n1-1 iterations and 'last' 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
blavp inc    +pcnt

/ Check to see if wrap around on circular queue during last sample value

mov    +xptr,r0
sub    $x,r0
cmp    r0,+wrap
bit    nowrap
mov    $x,+xptr
nowrap mov    +xptr,r0      / set r0 pointing to start of current packet
add    +vinc,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 'last' calculations
beq    last

/ Go to section of code that performs no calculations
bpl    chklist

/ Go to section of code that performs 'norm' calculations
jmp    norm

/ Perform last section of calculations
last  mov    +xptr,r2      / set counter to 'last'

/ Subtract oldest past sample from accumulated double precision sum
loopr sub    (r1),(r2)
sbc    3000(r2)           / make double precision subtraction

/ Replace past sample in 'x' with current sample
mov    (r0),(r1)+

/ Add current sample into accumulated double precision sum
add    (r0)+,(r2)
adc    3000(r2)           / make double precision sum

/ Move double precision 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    +b19h,r4

/ Round off after divide (ndb19 is set in C routine)

        cmr    r5,endlb19
        btl    nand
        inc    r4    / round up

/ Store background time average in second 256 words of m

        nand    mov    r4,776(r2)

/ The background time average will now be used to look up the
/ appropriate threshold in a table (tblat) which is pointed to
/ by bstr. This table contains threshold 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
/ procedure 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

        cmr    +oldtref,$3
        bst    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
        btl    $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,+xptr
        clr    +incr

/ Check if Pth iteration

        chbtl cmr    +pnt,+b19h
        bne    dard / go to decision routine

```

```

/ Reel at point

```

```

    cin    +rnt1
    br     dead

```

```

/ Reel from 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    +norm,r3
loopb   sub    (r1),r2
        sbe    3000(r2)
        mov    (r0),r1
        add    (r0),r2
        add    3000(r2)
        mov    3000(r2),r4
        mov    (r2),r5
        div    +bign,r4
        cmp    r5,+dlnsh
        bit    normd
        inc    r4
normd   mov    r4,776(r2)
        cmr    +oldref,43
        bsl    noshf2
        asr    r4
        bic    $1,r4
        add    +bstr,r4
        mov    (r4),r5
        asl    r5
        mov    r5,1776(r2)
        br     retn2
noshf2  asl    r4
        add    +bstr,r4
        mov    (r4),1776(r2)
retn2   sob    r3,loopb

```

```

/ Reel pointers

```

```

    mov    r1,+xptr
    mov    r2,+mptr
    add    +tenorm,+xnorm

```

```

/ Reel in main section of calculations

```

```

/ See comments at start of computational section for description
/ of a vector

```

```

dead    mov    $+a,r1
        add    $1,r0,r1      / r1 points to background time average
        mov    +xstr,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
        cin    r5
        mov    sp,+lmp1      / save stack pointer
        cin    +lar          / clear target flag
load    mov    (r1),r0      / put background time average into r0
        sub    (r2),r0      / subtract current sample value

```

```

/ Take the absolute value of difference
    bpl     r0s
    neg     r0s

/ Shift bit in r4 to make room for current threshold violation
/ information
    ror     r4, #1

/ Compare threshold with absolute value of difference of current sample
/ and background time average
    cmp     r0s, r10, r0
    bgt     nobis / test if no threshold violation

/ Set second bit to indicate threshold violation at current time instant
    bis     r2, r4

/ Mask off n divided by 3 most recent threshold decisions
/ 'mask' is set in 'C' routine
    nobis   bsr     +mask, r4

/ Move history into r0
    mov     r0, r4, #0

/ 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     r0, 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     r3, r3
    blt     init

/ Compare t to the sum of the number of threshold violations in the
/ three spatially adjacent cells which have occurred during the last
/ n divided by 3 packets
    cmp     +sm1, 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, sm1, r5
    init    sub     r3, loopd / go to next spatially adjacent cell in array

```

```

here    here    / save filler
here    mov     +tmp1.sp    / restore stack pointer
here    mov     +tmp2,r5    / restore r5
here    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 Newton's 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 (sen). The d parameter is assumed
/ to be in a nnnnn nnnnnn 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:  rgr     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 set threshold

```

```

          mov     r3,r0
          mul     +sen,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     +tblab
          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 MCR
/ (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    .set    r5,csz
        .mov    $200,r4
        mips:   /disable system interrupts

```

```

/ Test r-w of various registers

```

```

        .mov    $172410,r0    /load MCR address
        .mov    $52525,r1    /load test pattern
        .mov    r1,(r0) /write test pattern into MCR
        .mov    (r0),r2 /read MCR back
        .cmp    r1,r2 /check if error
        .beq    MCRok1
        .inc    +erMCR /set error flag
MCRok1  .com    r1 /try new pattern
        .mov    r1,(r0) /write into MCR
        .mov    (r0),r2 /read MCR back
        .cmp    r1,r2 /test if error
        .beq    MCRok2
        .inc    +erMCR /set error flag
MCRok2  .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 cleans appropriate bits
        .clr    r1

```

```

/ Set all possible bits in various registers

```

```

mov    $1,(r0)+
mov    $2,(r0)+
mov    $3,(r0)+
mov    $4,(r0)
mov    $172410,r0      /set r0 pointing to WCR
reset  /issue reset command (bus INIT)

/ Test if error bits are cleared in various registers

mov    (r0)+,+exec
bne    RSTer
mov    (r0)+,+exec
bic    $1,+exec
bne    RSTer
mov    (r0)+,+exec
cmp    $200,+exec
bne    RSTer
mov    (r0)+,+exec
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,+exec /set up expected status line value:
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),+lmp1 /read CSR
bic    $170777,+lmp1 /mask off status bits
cmp    +lmp1,+exec /test if error
beq    FCTok
inc    +erFCT /set error flag
FCTok  sub    $1000,+exec
sub    $2,r0
bpl    FCT /go to next set of f line values

/ Test DMA data transfers and end of load interrupts

mov    $-1000,$172410 /set up WCR
mov    $+buf1,$172412 /set up BAR

/ Put something interesting in ybuf1

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

/ Enable system interrupts

clr    r4
mle-

/ Hit maint. and ie bits

```

```

        mov     $172414, *flag
        bne     $400, *flag2414      / hit cycle repeat

/ Go into wait loop
        clr     *errorpec
wait:    inc     *errorpec
        bne     wait

/ Test how things went
/ Check for interrupt
        tst     *flag
        bne     INTok
        inc     *errorINT /set error flag
INTok:   mov     *bus1, r0
        mov     $400, r1
        TRN     mov     (r0), r2
        cmp     r2, (r0) +          test if proper word transferred into memory
        bne     TRNerr
        sob     r1, TRN
        br      TRNok
TRNerr:  inc     *errorTRN /set data transfer error flag
TRNok:   reset
        jmp     orcl

/ 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 processor 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)+
jnb 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:
moub (r3),r4    /set low order byte
add r4,(r2)     /compute low order checksum
inc r3         /go to high order byte
moub (r3),r4    /set high order byte
add r4,2(r2)    /compute high order checksum
inc r3         /go to next word
jnb r0,loopc
add $4,r2       /move pointer to store next 11 checksums
halt

```

0215501 BARok1	0246621 L20018	0416421 L20127	0252701 L283
0215661 BARok2	0250061 L20020	0423601 L20128	0253141 L286
0216121 BARok1	0251341 L20022	0424041 L20129	0253741 L287
0216301 BARok2	0252301 L20024	0424301 L20130	0255121 L292
0217521 FCT	0253261 L20026	0424541 L20131	0255501 L293
0220021 FCTok	0254321 L20028	0425001 L20132	0577501 L3
0221261 INTok	0253101 L20029	0425241 L20133	0266601 L300
0577441 L1	0260761 L20032	0425501 L20134	0256401 L301
0577701 L10	0261521 L20034	0230441 L202	0256701 L302
0222401 L10000	0262261 L20036	0230541 L203	0266041 L303
0576461 L10002	0263021 L20038	0230641 L204	0257001 L305
0576721 L10004	0263561 L20040	0230741 L205	0260621 L307
0237021 L10005	0264561 L20042	0231041 L206	0257521 L308
0236721 L10006	0256001 L20045	0231141 L207	0260161 L314
0241641 L10007	0305621 L20047	0231241 L208	0260501 L316
0241341 L10008	0310561 L20049	0231341 L209	0265121 L318
0243321 L10009	0311721 L20051	0231641 L210	0261361 L319
0307741 L10010	0312321 L20053	0232141 L213	0261101 L323
0314021 L10011	0314441 L20055	0233141 L217	0262121 L325
0321121 L10012	0316161 L20060	0233201 L219	0261641 L329
0323321 L10013	0316661 L20062	0233361 L220	0262661 L331
0323221 L10014	0316741 L20064	0234261 L222	0262401 L335
0326361 L10015	0317421 L20066	0234361 L223	0263421 L337
0330261 L10016	0321461 L20068	0234021 L225	0263141 L341
0333041 L10017	0323741 L20070	0235341 L231	0264161 L343
0577201 L10019	0326721 L20072	0236441 L233	0263701 L347
0577361 L10021	0330621 L20074	0236101 L234	0264761 L350
0577721 L11	0331421 L20076	0236401 L236	0264641 L353
0577741 L12	0333401 L20078	0237101 L238	0266441 L357
0577761 L13	0334201 L20080	0237421 L240	0267401 L361
0222121 L175	0336421 L20082	0237661 L242	0270021 L365
0224441 L179	0340201 L20083	0240041 L243	0275101 L370
0224501 L180	0350201 L20085	0252121 L246	0305341 L397
0225741 L184	0351501 L20087	0242141 L247	0305401 L398
0226521 L185	0353101 L20089	0242101 L249	0306021 L399
0226761 L188	0354461 L20091	0241001 L250	0577521 L4
0227041 L189	0356061 L20093	0241141 L251	0306701 L402
0227121 L190	0364161 L20095	0241721 L252	0306761 L403
0227201 L191	0365241 L20097	0242501 L254	0310101 L406
0227261 L192	0365201 L20099	0242741 L255	0310141 L407
0227341 L193	0230341 L201	0247421 L256	0310761 L408
0227441 L194	0365661 L20101	0245161 L259	0312101 L411
0227541 L195	0372761 L20103	0245041 L260	0314161 L419
0227641 L196	0374401 L20105	0245001 L261	0314221 L420
0227741 L197	0374721 L20107	0246261 L263	0314641 L421
0577461 L2	0400461 L20108	0246341 L264	0316401 L425
0230241 L200	0402521 L20110	0246221 L265	0317121 L427
0222061 L20001	0406401 L20112	0246161 L266	0321261 L435
0223441 L20003	0406341 L20114	0247021 L269	0321321 L436
0223741 L20005	0410061 L20115	0250441 L272	0321661 L437
0223341 L20006	0404001 L20117	0250261 L273	0323461 L442
0233541 L20008	0407521 L20118	0250701 L276	0323521 L443
0234721 L20010	0411561 L20120	0251101 L277	0324141 L444
0240561 L20012	0412621 L20122	0251261 L278	0325301 L447
0244001 L20014	0413121 L20124	0251561 L279	0326521 L450
0245321 L20016	0415041 L20126	0252161 L282	0326561 L451

032712L L452	040320L L595	041612T +alarm	044462D +c21st
033042L L457	040676L L596	062010B +aldrsab	032774T +c2srv
033046L L458	040444L L598	051640D +aldsl	143670B +c2str
033102L L459	040672L L599	144030B +almfls	062040B +cc1
033320L L467	057756d L6	143264B +arlr	062026B +cc2
033324L L468	040534L L600	024014T +asc1on	143742B +chksm1
033360L L469	040720L L608	143714B +atdbin	143744B +chksm2
033614L L477	040772L L610	143704B +atdoct	143746B +ck1
033610L L478	041454L L617	035200T +atod	050430D +ck1loc
033572L L480	041244L L621	052700D +atod1	143756B +ek2
033620L L481	041300L L622	052746D +atod2	050440D +ck21oc
033626L L482	041424L L624	053044D +atod3	050544D +ckcrin1
033634L L483	041646L L635	143740B +atodbyt	050630D +ckcrin2
033646L L484	041636L L638	041142T +atodec	051362D +elcorn
033704L L486	041652L L639	035466T +atodins	043302D +cmdno
033712L L488	041660L L640	034774T +atodtst	043336D +cmds
033720L L489	042010L L643	041460T +atooct	143600B +command
033726L L490	041714L L644	143604B +audalrm	043500D +orlf
034014L L494	041740L L645	043000T +bds	020334T +crunch
057754d L5	041732L L646	043476D +bell	043442D +ctrh
035174L L504	041766L L648	143562B +bfrt11	043460D +ctrx
035024L L505	042206L L659	143564B +bfrt12	143576B +dctares
035044L L507	042212L L660	143566B +bfrt01	046532D +dctar2
035154L L509	042354L L665	143570B +bfrt02	046442D +dctarr
035262L L514	042400L L667	143606B +bisd	143620B +dctstrn
036172L L527	042424L L669	143634B +bisdstr	044634D +ddl1
03612L L528	042450L L671	132070B +bign	045426D +ddl10
036452L L530	042474L L673	143640B +bignstr	045526D +ddl12
036002L L531	042520L L675	136076B +bisp	045650D +ddl14
036152L L532	042544L L677	143644B +bispstr	045770D +ddl16
036136L L533	057760d L7	047656D +binloc	046110D +ddl18
037062L L539	057763d L8	043504D +bksp	046230D +ddl19
036422L L540	057766d L9	043372D +blank	044726D +ddl2
036464L L542	021730L RSTer	046422D +blnk	044734D +ddl3
036636L L551	021734L RSTok	042040T +box	045006D +ddl4
036624L L552	022136L TRN	051400D +box1	045124D +ddl5
036676L L554	022150L TRNer	051420D +box2	045166D +ddl6
036722L L556	022154L TRNok	051430D +box3	045306D +ddl8
036766L L557	021512L MCRok1	051440D +box4	030152T +ddlist
036750L L558	021530L MCRok2	051450D +box5	043510D +ddloc
037004L L561	052034D +BARer	042740T +bss	043776D +ddlst
037050L L562	057642D +CKSH1	132074B +bstr	030472T +ddsrv
037032L L563	057644D +CKSH2	137122B +btab	023452T +decode
040116L L571	052110D +DBRer	143572B +btent1	046244D +disp2
037302L L572	051740D +DMA11	143574B +btent2	046604D +disp3
037344L L574	056736D +Err	144034B +buf	046670D +disp3b
037454L L575	052340D +FCTer	060002B +bufnum	034530T +display
037506L L577	052444D +INTer	143612B +cl	030270T +dlst
037566L L579	056706D +Noer	030440T +cllist	043612D +dlloc
037604L L580	052164D +RSTer	043746D +clloc	044222D +dlst
040016L L581	052604D +TRNer	044550D +cllist	021454T +dma
040052L L582	051760D +MCRer	033252T +clsrv	054010D +dma1
040034L L583	043056T +cleanu	143676B +clstr	054054D +dma2
040076L L585	136104B +a	143614B +c2	054152D +dma3
041012L L592	033734T +addrreal	030406T +c21st	054206D +dma4
040256L L593	047704D +adtst	043716D +c2loc	054310D +dma5

054362D +dma6	047140D +line2	050072D +pdsdis2	057156D +sproc
054430D +dma7	047216D +line3	036176T +pdsinc	137116B +sen
054502D +dma8	047274D +line4	035562T +pdsist	051506D +star2
042564T +dmains	047322D +line5	132066B +plast	051460D +star21
022162T +dmaint	047352D +line6	136100B +pnorm	020000T +start
042230T +dmals	047402D +line7	030236T +pplist	143272B +stinst
042170T +dmatlst	047446D +line8	043564D +pploc	043060D +string
143270B +drucsr	046760D +line9	044136D +pplst	043202D +tab
062022B +dshft	033530T +lsrv	031354T +pssrv	137106B +tar
032066T +dsrv	132076B +m	050236D +pmdisl	051534D +tarsl
020310T +dummr	022174T +main	050264D +pmdis2	062020B +tbsd
060000D +edata	137110B +mask	050320D +pmdis3	037066T +termins
144046B +end	050202D +makon	040122T +pmins	036372T +termist
020122T +endek	050150D +kuvor	055226D +pnom1	050140D +test
143124B +erBAR	132062B +mrtr	055274D +pnom2	021316T +thresh
143126B +erDBR	132064B +nl	055370D +pnom3	030354T +tlist
143134B +erFCT	030322T +nlist	055444D +pnom4	043670D +tloc
143136B +erINT	043642D +nloc	050470D +pnom4	044400D +tlist
143132B +erRST	044310D +nlist	037252T +pnomlst	137104B +tmap1
143140B +erTRN	030204T +nnlist	034256T +putdsp1	062014B +tmap2
143122B +erWCR	043536D +nnloc	026702T +puttlv1	137114B +tmap3
144016B +errent	044054D +nnlist	026744T +puttlv2	136102B +tenorm
144066B +erriok	030746T +nnsrv	051146D +trani	062012B +tref
050450D +ex1loc	052062D +noBAR	051062D +trancnt	062050B +trefml
050460D +ex2loc	052136D +noDBR	050714D +tramer	032600T +tsrv
143766B +eck1	052402D +noFCT	054552D +trani1	143276B +ttribut
143776B +eck2	052526D +noINT	054620D +trani2	143314B +ttrcont
043040T +exit	052252D +noRST	054706D +trani3	143310B +ttrifrs
143130B +expec	052642D +noTRN	055012D +trani4	143312B +ttrifro
043424D +f10	052006D +noWCR	055062D +trani5	143274B +ttristat
043406D +f9	051200D +noran	055166D +trani6	022246T +ttrvin
060000B +flas	051610D +notarl	041016T +trains	046724D +work
143732B +flines	143266B +np	040226T +tranlst	132056B +wmap
047774D +flnloc	043062D +nel	132072B +trdrsh	062056B +x
062024B +fslat	032254T +nsrv	050766D +ttech	062054B +xptr
022172T +halt	062016B +ntbksra	046336D +seelst	061004B +ybuf1
042014T +help	047674D +occlloc	025546T +send	060004B +ybuf2
047564D +help1	144026B +oldalifs	055540D +ser1	132060B +yiner
056762D +hithit	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 +rcnt	056162D +ser6	177770a b
056610D +hitlst	053116D +pds1	056242D +ser7	177770a b
144032B +holdcha	053734D +pds10	056344D +ser8	177770a b
143632B +holdent	053762D +pds11	056446D +ser9	177770a b
047514D +home	053214D +pds2	046576D +sk21	177770a b
027006T +initial	053326D +pds3	047040D +skip	020546t blava
062006B +intfla	053402D +pds4	137120B +smd	177760a brtecht
041664T +inout	053502D +pds5	143650B +smdstr	177752a c
056700D +ium	053530D +pds6	143616B +sml	177762a c
143142B +junkln	053624D +pds7	143610B +smn	177770a c
046736D +learn	053660D +pds8	143648B +smnstr	177754a c
047054D +linela	053706D +pds9	137112B +smt	177742a c
047106D +line1b	050004D +pdsdis1	143660B +smtstr	177770a c

17770a c	17776a i	177760a shift	032600t -tsrv
177762a c	177764a i	177756a stackst	022246t -ttrn
177754a c	177770a i	020460t stinc	
177766a c	177770a i	021270t tarsa	
177744a carrv	177756a i	020516t update	
020772t chklst	177770a i	022106t wate	
177762a cksler	021274t init	020232t vbf2	
177760a cks2er	022174a inced o	033734t -address	
020420t ckst	000000a inced-in	041612t -alarm	
020252t cmpt	177770a j	024014t -aselon	
177770a cntr	177766a j	035200t -atod	
020104t crel	177764a j	041142t -atodec	
020266t crn	177770a i	035466t -atodins	
020066t csu	177764a k	034774t -atodlst	
043040a cuevit o	177764a k	041460t -atodocl	
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 -domains	
177756a dfls	021204t loopd	042230t -dmats	
177766a divide	177760a m	042170t -dmatsl	
177762a divide	177764a m	032066t -dsrv	
177760a ercnt	106404a mtes	042014t -help	
177754a erflg	177762a n	027006t -initial	
177762a erflg	021230t nobis	041666t -inout	
177756a erloc	020272t nocrn	033530t -lsrv	
177766a excks1	000240a nor	022174t -main	
177764a excks2	020700t nord	030322t -nlist	
000001a exit	021010t norm	030204t -nnlist	
043056a fakeu o	021054t nornd	030746t -nnsrv	
177766a fln	020740t noshf1	032254t -nsrv	
177750a frac	021114t noshf2	023252t -pance	
000000a halt	020576t nouwar	036176t -pdsinc	
021302t here	000004a num	035562t -pdsist	
177770a i	000004a num	030236t -pelist	
177764a i	021214t pos	031354t -psrv	
177770a i	177746a powr	040122t -prmins	
177770a i	177750a ptin	037252t -promist	
177770a i	177746a ptoul	034256t -puidsp1	
177770a i	000004a r1	026702t -pullty1	
177770a i	000003a r2	026744t -pullty2	
177766a i	000002a r3	041016t -ramins	
177770a i	000005a reset	040236t -ramist	
177770a i	020752t retrn1	025546t -send	
177770a i	021126t retrn2	037066t -termins	
177766a i	000002a rli	036372t -termist	
177764a i	020140t setfls	030354t -tlist	

MISSION **of** **Rome Air Development Center**

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C³I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

DA
FILM
2-