(NASA-CR-163486) REAL-TIME OPERATING SYSTEM FOR A MULTI-LASER/MULTI-DETECTOR SYSTEM (Rensselaer Polytechnic Inst., Troy, N. Y.) 120 p HC A06/MF A01 CSCL 20E
N80-30726
Unclas
G3/36 28477

RENSSELAER POLYTECHNIC INSTITUTE
KNOWLEDGE AND THOROUGHNESS
1824

Rensselaer Polytechnic Institute

Troy, New York 12181

RPI TECHNICAL REPORT MP-65

REAL-TIME OPERATING SYSTEM FOR A MULTI-LASER/MULTI-DETECTOR SYSTEM

by

Gary Coles

A Study Supported by the

National Aeronautics and Space Administration

Grant NSG-7369

School of Engineering
Rensselaer Polytechnic Institute
Troy, New York

July 1980

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The first hazard detection system used on the Rensselaer Mars rover was the one laser-one detector system. This system is reviewed briefly with respect to the hardware and software subsystems, the operation, and the results obtained.

Recently, a multidetector scanning system has been designed to improve on the original system. Interactive support software has been designed and programmed to implement real time control of the rover or platform with the new elevation scanning mast. The formats of both the raw data and the post-run data files have been selected. In addition, the interface requirements have been selected, and some initial hardware-software testing has been completed.

PART 1

INTRODUCTION

Our knowledge of the solar system has increased dramatically over the past decade as our mode of planetary exploration has approached the planet surface. The rate of new discoveries has accelerated in the progression from Earth-based observation to orbital or fly-by missions, to landers, and finally to the ultimate manned missions. As these missions extend farther from Earth, it becomes prohibitively expensive to send up a manned mission. One possible alternative is some type of unmanned autonomous rover which has the capability to cover a great distance over the planet's surface. In this way it could visit a number of interesting scientific sites over an extended period of time.

An important part of such a rover is its path selection and hazard avoidance control system. Due to a large communication delay time between Earth and any other planet (except possibly the Moon), Earth-based control other than on the macro level would be out of the question. Therefore the vehicle control system would have to be self-contained and highly reliable.

The Mars rover group at Rensselaer has been trying to develop such a vehicular control system. These studies include simulation and more recently, real time control on a prototpye rover.

Results indicate that such a path selection system should be broken down into two to four distinct levels. Long-range goals could be planned to about the kilometer range using photographs from either

Earth or satellite observation. Medium range paths in the tens of meters could be selected onboard the rover using some kind of range-finder or television interpretation technique. Finally some kind of short-range technique must be used within about three meters of the vehicle to detect all hazards too small to be resolved in the longer range planning.

This report will discuss the organization and design of the real time support software used to implement the short-range path selection system on the Rensselaer Mars rover. Besides implementing real time control, this software provides for easy program development. Programs developed on the simulator can be run under real time control with only minor modifications. Also, all the raw data provided by the rover is saved for post-run analysis.

PART 2

THE RPI MARS ROVER

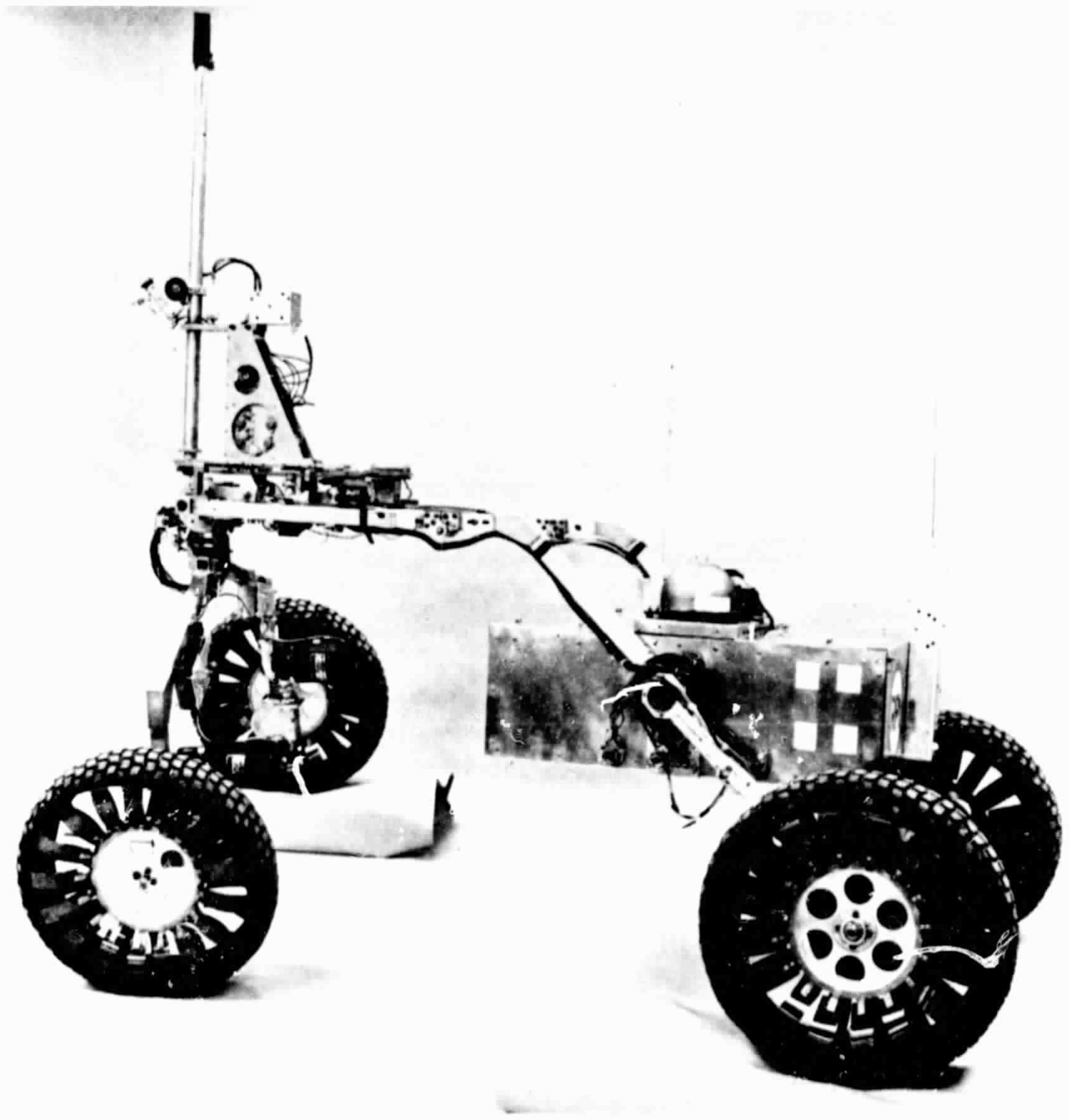
A. Rover Description

A 0.5 scale prototype Mars rover was constructed at Rensselaer to test hardware features and realtime control software (see Figure 1). A short range hazard detection system has been implemented to enable closed loop realtime path selection testing. Note that another group at RPI is working on medium range path selection techniques using simulation.

The payload of the rover consists of a heading gyro, a pitch-roll gyro, an electronics section, and three automobile batteries used to power the motors and electronics. The electronics section includes the telemetry transmitter, the speed and turning controller, the scanning mast controller, and the data acquisition and telemetry controller. On this system, the pitch-roll gyro was not used.

The front axle of the rover is rigid and pivots directly under the mast. There are currently 15 steering positions from -90° to +90° in 12.86° increments. The immediate steering angle is read from a linear potentiometer connected between the frame and axle. Besides pivoting about a vertical axis, the front axle can also pivot about a heading or roll axis. This front axle roll angle is also read from a linear potentiometer, but it is not used in this system.

The drive system of the rover consists of four motors, one on each wheel. Each motor has a tachometer connected to it to obtain wheel speed data. Steering the rover is accomplished by changing the speed of the four wheels such that a smooth turn is achieved while



RENSSELAER AUTONOMOUS ROVING VEHICLE

Figure 1

maintaining approximately constant velocity. There are also two motors which allow the front and rear wheel struts to be raised or lowered. This gives the ability to raise or lower the payload, but it is not used during realtime control.

B. The One Laser-One Detector System

The Rensselaer Mars rover uses a laser triangulation scheme for its short range hazard detection system (see Figure 2). The laser is a solid state GaAs laser diode which has ten watts peak power output at 904 nanometers. The beam is collimated and directed straight up the mast. A mirror is mounted on top of the mast and directs the beam at an adjustable angle toward the ground. A silicon PIN diode is used as a receiver. It is mounted part way up the mast and focused in a known cone of view toward the ground. The beam and detector cone are adjusted to intersect at ground level such that the beam will be detected when falling on terrain between about ± 30 cm from level (see Figure 3). This intersection is adjustable and is typically set at 1.5 meters from the mast axis.

The entire mast then oscillates back and forth scanning a 140° field of view. During each sweep the laser fires at 15 azimuth angles, one per 10° increment, and centered on the steering heading (see Figure 4).

The laser data is composed of one 15 bit word per scan, one bit per azimuth angle (see Figure 5). The information provided by each bit is of a go - no go nature where a one or "good" return specifies a safe azimuth, while a 0 or "bad" return specifies a possible hazardous azimuth.

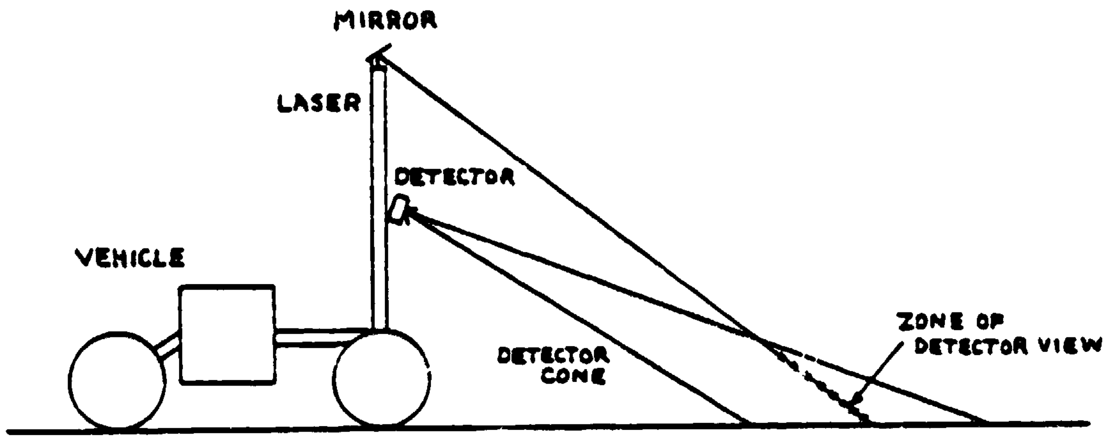


FIGURE 2. One Laser - One Detector System

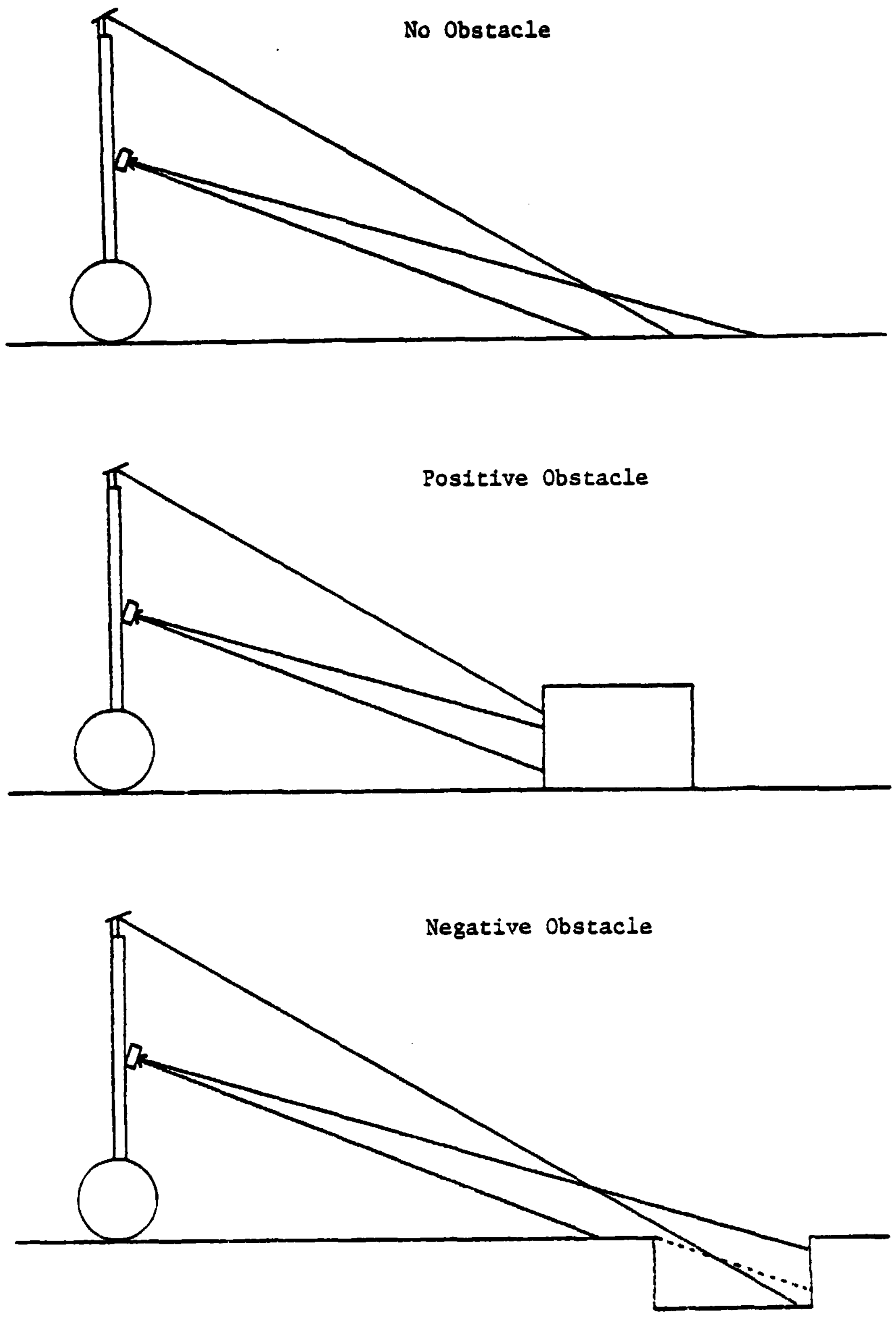


FIGURE 3. Laser Triangulation Concept

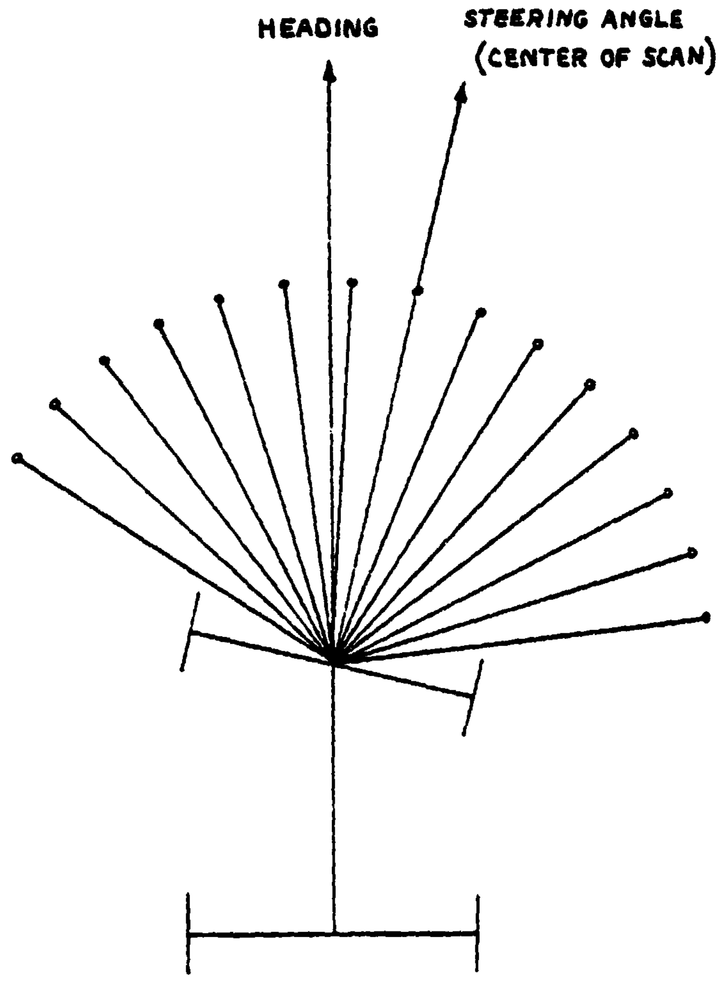
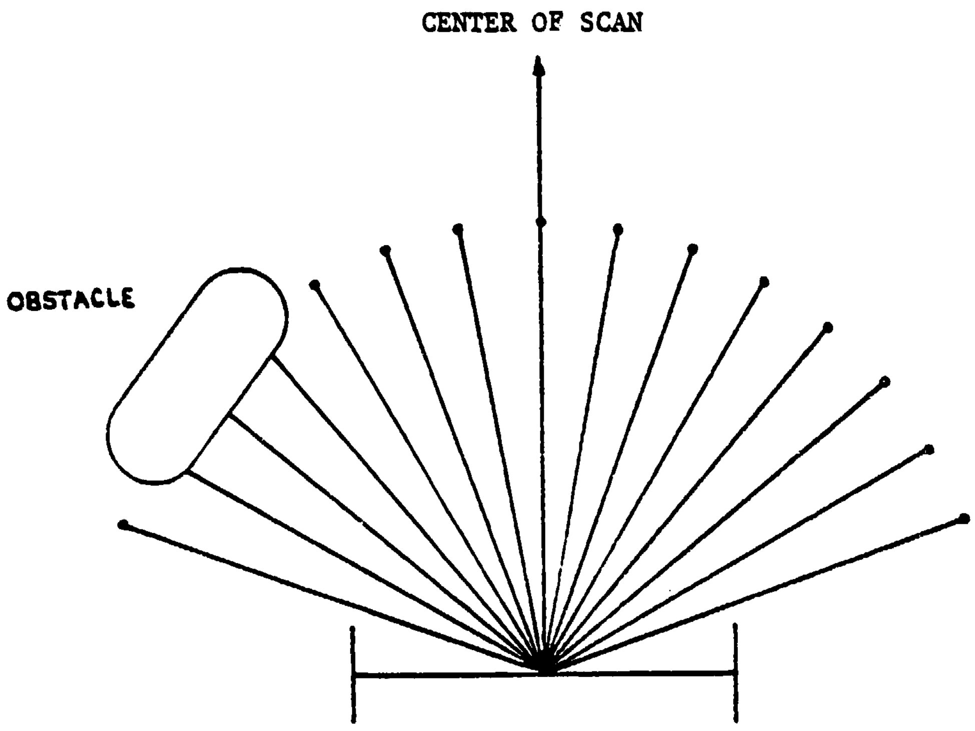


FIGURE 4. Laser Azimuths



```
15                              0
|X 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1|
```

FIGURE 5. Laser Data Word

C. The Varian-Idiiom System

A Varian 620i minicomputer is used to control the rover. It is a 16 bit computer with 32K words of core memory. Included in this system is a Tektronix terminal, a 10 megabyte cartridge disk drive, two 800 bpi tape drives and an Idiiom graphics display. The Varian is of the 1960's era and therefore does not have virtual mapping or hardware floating point. There are three main registers: the A register, used as a general accumulater, the B register, used for indexing and as an extension of the A register for certain operations, and the X register, used for indexing. As an example of performance, a 16 bit by 16 bit multiply with one operand in register takes between 18 and 20 μsec.

The disk memory unit is used to store off-line programs; no runtime overlays are used. Raw data is saved on one of the magnetic tape units for post run analysis and the Idiiom terminal is used to display important runtime parameters as well as a map of the current rover position. The Idiiom also provides a 60 Hz realtime clock interrupt which is used for timing.

The rover is linked to the Varian by a two-way telemetry transmitter. A receiver at the Varian side performs error correction and then inserts the data into memory using DMA (Direct Memory Addressing). The data is always put in the same table and each new value always overwrites the last respective value. In this way, the most recent value of any desired data word is always found in the same respective location. The maximum rate for DMA is 202,000 words per second.

A command to be sent to the rover from the computer is one word long. An OTA (output from A register) instruction is used to give this data to the Varian interface which then sends the command.

Another communications link to the rover is the remote control box. This box disables and overrides the computer control, and is used to provide manual control of the vehicle. It is useful in positionir g the rover and for gaining emergency control.

D. Realtime Software

The only language currently available on the Varian computer is assembler and therefore all of the realtime programs are written in assembly language. Also, although previous descriptions of this software mention the use of external interrupts, they were not used due to hardware problems.

The main object of the realtime software is path selection. Hazards must be identified and avoided, and the entire system state must be saved for later analysis. Most of the programs developed earlier for this system (see Reference 1) were left intact except for a few modifications. The major changes were in the path selection routine as expected.

The first workable idea to be implemented for realtime testing was termed "path-blocking," conceived earlier by M. Krajewski (see Reference 2). Path-blocking involved the buffering of bad bits (hazards) in the laser data word. Specifically, four bits on both sides of any bad bit in the laser data word were set bad to buffer any obstacles. A clear path consisted of any four contiguous good bits. The clear path closest to the desired heading became the steering angle. If a clear path could not be found then the rover would scan all possible steering angles by turning its front axle, and thus its center of scan, to $-70°$ and back to $+70°$ looking for a clear path. If one could not be found then the rover would halt because back-up capability, although used in simulation, was not implemented.

To remember obstacles behind the line of sight, a laser data memory queue was used. This first-in first-out queue was of variable length, and new entries were shifted left or right according to the steering angle. All of the elements were logically "anded" together with the newest scan before looking for a clear path.

This early technique worked but it proved to have some flaws. Although it was able to keep the front wheels clear, the back wheels frequently clipped the obstacles on passing. This could be improved by increasing the length of the laser memory or increasing the width of the path-blocking buffer to five bits. Unfortunately, both of these solutions tended to make the overall system very conservative. It was determined that path-blocking, although effective for the front wheels, was ineffective for the back wheels.

The solution to this problem was a two part path selection system, one dealing with the front wheels, and another for the back wheels. This algorithm called track-and-turn (TRKTRN) was pioneered by T. Sadeghi and continued by P. Dunn (see Reference 3).

Since path-blocking seemed to work, it was kept for the front wheels. The back wheels used a new technique based on the position of the obstacle and the geometry of the rover. Given the geometry of the rover, the possible rear wheel trajectories could be calculated for any steering angle. Therefore, a formula was developed which used the position of the obstacle to yield the necessary steering angle. Finally this steering angle would have to be truncated to one of the possible steering angles. The last step was to combine the front and rear wheel constraints and choose the desired steering angle.

The obstacle memory for the TRKTRN system was more complicated than the previous memory. The path-blocking laser memory was retained for the front wheels. However, the back wheels formula required the locations of the obstacles in the planet frame to be saved.

Other realtime software included routines to decode vehicle state data, to send commands to the rover, to keep track of the vehicle position, to update the Idiiom display, and to save the system state for later analysis (see Reference 4). Another program was written to run off-line on the IBM 360 computer to analyze this saved runtime data.

E. Results

The results of the lab tests were very encouraging. The rover was able to successfully maneuver through almost every obstacle pattern that was given to it. The few times it failed can be attributed to the inability of this simple path selection system to resolve a tight situation as being passable.

In the field tests, the real problems showed up as expected. A little bit of pitch and roll made the system even more conservative. A larger but still passable pitch or roll was interpreted as being hazardous. All in all, the system performed quite well. The rover almost always found its target although not always by the most direct route.

Most of the inadequacies of the one laser-one detector system can be traced to its inability to interpret and adapt to terrain containing slopes. However, this system still has some usefulness. Not all ideas have been tried to reduce its conservatism in the lab tests.

Overall, the rover was plagued by many problems. Most of the mechanical problems were due to the fact that the rover was not designed for the weight load which it acquired over the past few years. This caused some gears, shafts and other structural members to fail without warning. The electrical system was plagued by a chronically bad wheel speed controller which caused the wheels to move at slightly different speeds and the front axle to oscillate. This put more stress on the rover's structure. Finally, the software always seemed to contain some hidden bugs characteristic of large programs written in assembly language.

PART 3

THE ELEVATION SCANNING SYSTEM

A. General Description

The necessary solution to the pitch and roll problem of the one laser-one detector system is to come up with some system which not only provides range data but also height data. This would provide the information necessary to interpret slopes.

Since the triangulation concept worked it was decided to go with a modification of the one laser-one detector system. This new system is known as the multi-laser multi-detector scanning mast (see Figure 6 and Reference 5). The major components of the ML-MD system are the spinning mirror scanner, the detector array, and the controller electronics.

It was necessary to build a new mast which could support the heavier weight of the new system components. This mast rotates counter-clockwise at a preselected rate, normally about 0.5 revolutions per second, and is electrically connected to the chassis by slip rings. With two seconds between every scan, it is not possible to ignore any scans, thus limiting the realtime processing to under two seconds.

A new laser diode was chosen which could meet the new speed and power requirements (see Reference 6). It has a 10 KHz maximum pulse rate and an output of 100 watts. New collimator optics were designed to take full advantage of the diode's output power. An eight-sided mirror with a motor and positional encoder is mounted on top of the mast. With this arrangement the laser beam can be deflected to the

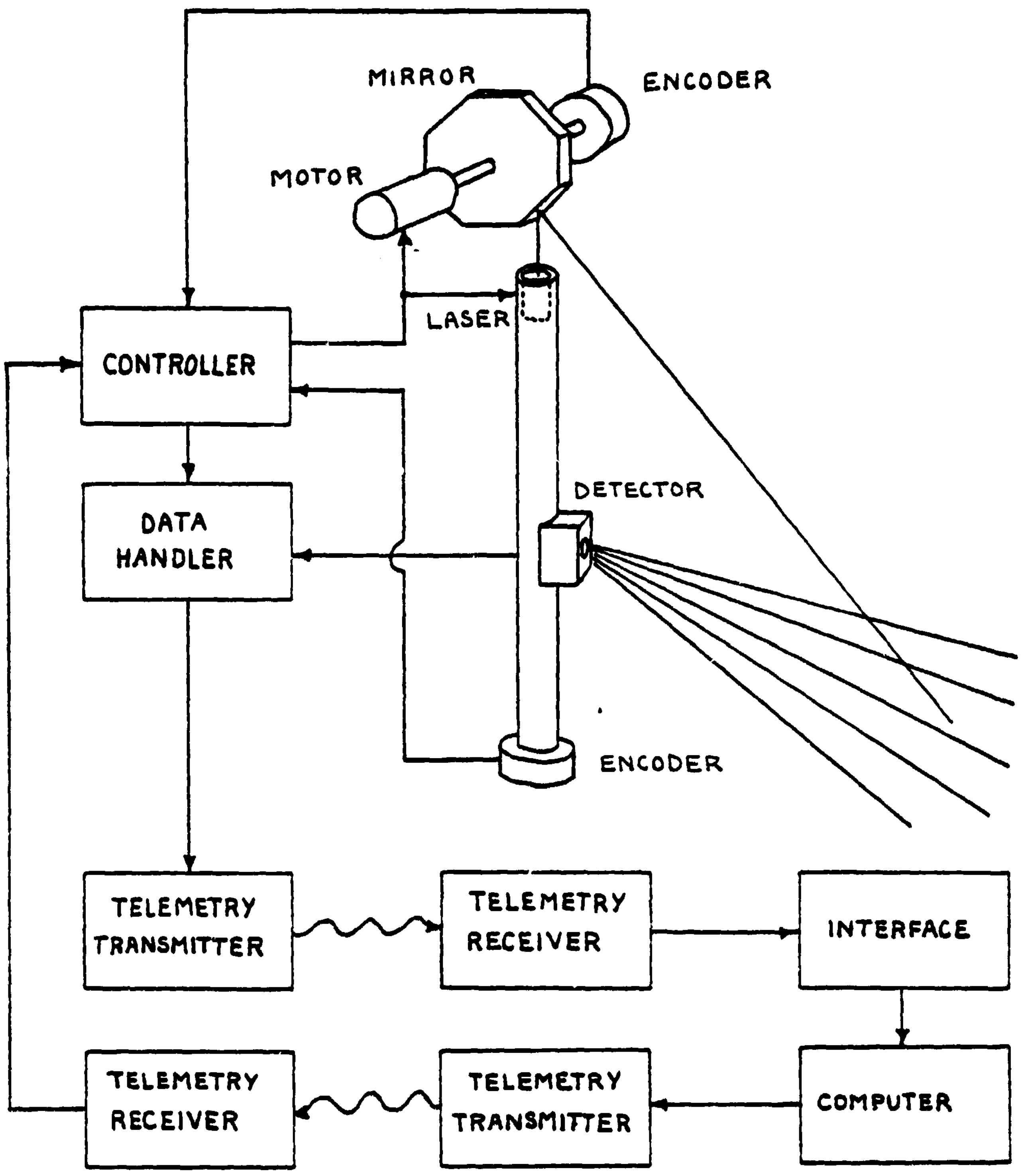


FIGURE 6. Elevation Scanning Concept

ground at an elevation angle selected by the rotational angle of the mirror. This is used to simulate multiple lasers firing at a number of preselected angles (see Figure 7).

The multi-detector receiver is composed of a linear array of photodiode elements. Currently two such devices are being examined for this use. One is a 20 element photodiode chip and the other is a 1024 element charge-coupled photodiode shift register chip. The 20 element chip receiver is known to work, but has the disadvantage of a limited number of receiver cones with non-alterable cone angles. On the other hand, the 1024 element chip has a greater resolution, and cone angles can be altered by assigning a different number of elements per receiver. Unfortunately the 1024 element chip has some noise and sensitivity problems which require more investigation. One possible method to increase the resolution of the 20 element chip receiver is to use two of them; however this creates problems in the design of the receiver optics.

Since the 20 element detector will probably be the first one used, it can be seen that the cone angles and sizes will be fixed by the optics selected. The laser elevation firing angles, however, are alterable and will be selected by a PROM (Programmable Read Only Memory) in the controller. In addition, the number of shots fired per azimuth, the number of azimuths, and the azimuth angles are also PROM selectable. At this time, the number of azimuths and the number of laser shots per azimuth must have a product less than or equal to 1024 (e.g., 16x64, 32x32 and 15x20). The controller must also insert the proper flags (e.g., end of azimuth) into the data stream to synchronize

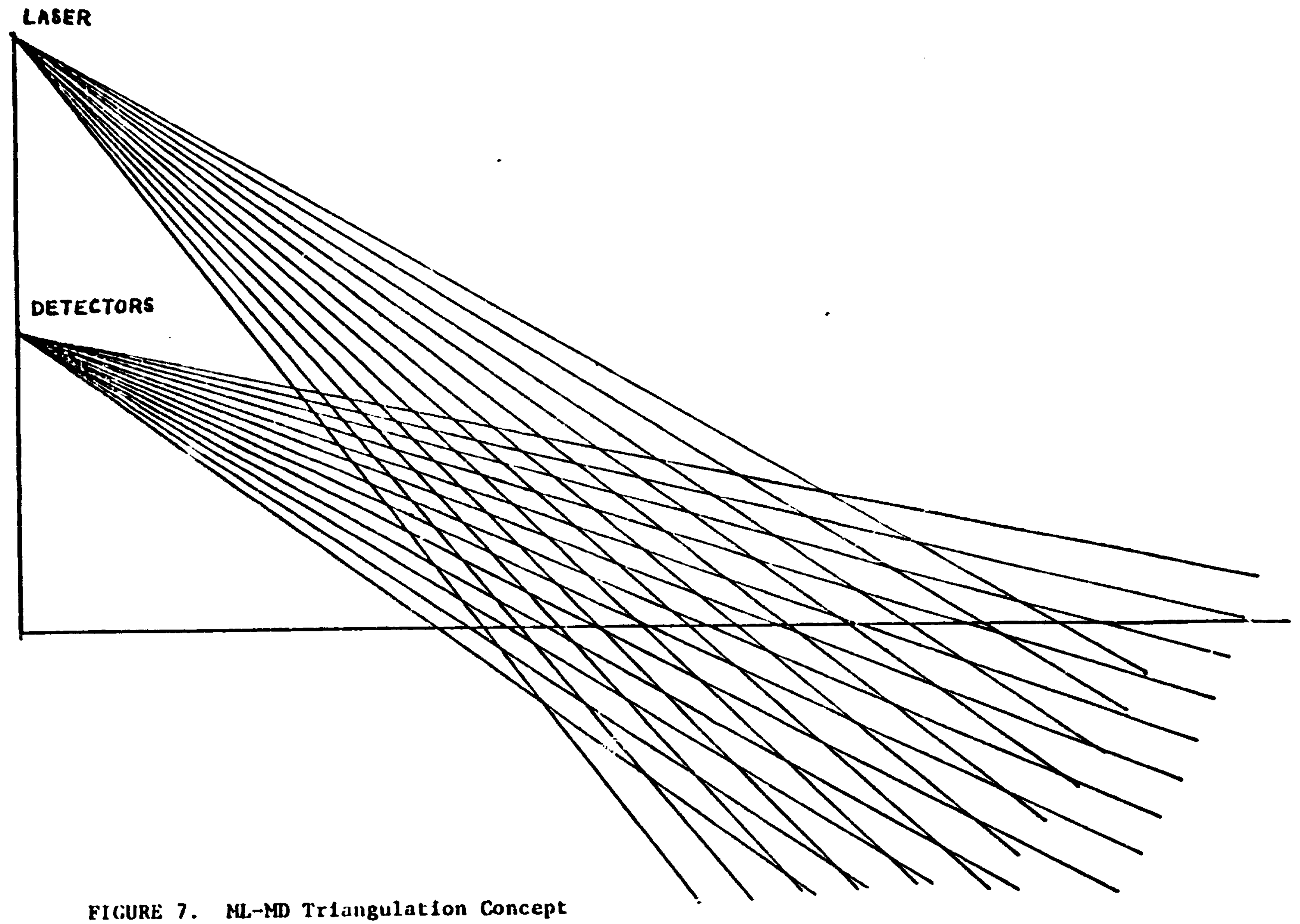


FIGURE 7. ML-MD Triangulation Concept

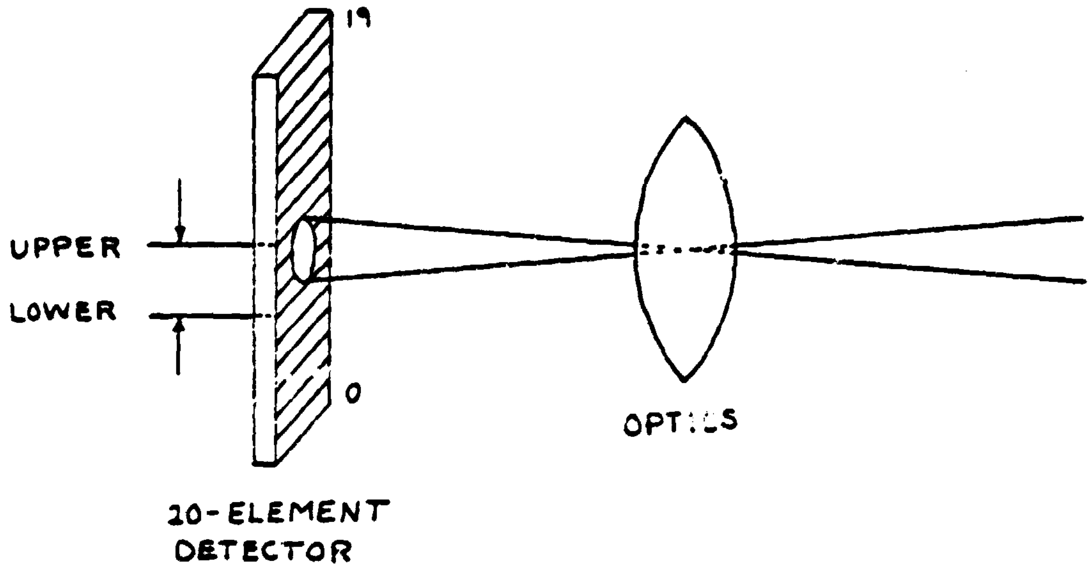
the data with the scan.

It is obvious that the elevation scanning system carries a great deal more information than the one laser-one detector system. Each laser data return is not a go - no go flag any more, it is a word specifying which receiver cone, if any, sensed the laser shot. Also, since the optics makes it possible for more than one receiver element to see the shot, each laser data return word contains the uppermost and lowermost cones to sense each shot (see Figure 8).

Besides the scanning system, other rover systems were upgraded. A new wheel-speed controller was designed which uses a Motorola M 6800 microprocessor to provide the reliability of digital control. A new telemetry system was necessary because of the greater data rate required by the new scanning system. And finally there were many modifications made to the electrical and mechanical subsystems such as rewiring and wheel strengthening.

B. The Prime Computer System

With the new scanning system, it was obvious that the Varian computer would not be able to perform realtime control very easily. The amount of raw data alone is about 1000 times more than before. Since the Varian would limit the complexity and precision of any new algorithms, it was decided to look for an alternative. The Image Processing Lab's Prime 500 computer was chosen as the best alternative. The Prime 500 is a new machine delivered in January 1979. It is a minicomputer with 32 bit internal architecture, hardware floating point and a very powerful instruction set implemented in microcode. As an example of performance, a single precision floating point multiply



15 … 10	9 … 5	4 … 0
0 0 0 0 0 0	UPPER	LOWER

FIGURE 8. Multidetector Data

takes about 4.0 μsec.

Besides having 512K bytes of main memory, virtual mapping is used to provide each user with up to 32 M bytes. The system includes two 80 megabyte disk drives, a mag-tape drive, a Versatec printer-plotter, and a number of terminals of which one is a dedicated operator's console. The Prime operating system is written for a timesharing environment servicing up to 63 terminals. Software includes FORTRAN IV, BASIC, Prime assembler, and a very powerful filing system (see Reference 7).

The ability to use higher level languages provides the benefit of easy program development and debugging. An added benefit is the direct compatability between programs developed for the simulator and for realtime.

The Prime 500 will shortly be replaced by a Prime 750 which is much faster. No instruction execution times are available as yet, but the Prime 750 includes a high-capacity cache memory, an instruction prefetch unit and a high speed floating point unit.

C. The New Rover Interface

To communicate with the rover it became necessary to build a new computer interface. Unfortunately, this was one of the things about the Prime that was most troublesome. What was to be a reasonably simple design task turned up some problems which are still as yet unsolved.

For the hardware part of the interface, a general purpose interface board (GPIB) was purchased from Prime. The GPIB enables the use of programmed input-output (PIO), standard and vectored interrupts, and a set of direct memory functions (DMX or DMA, DMC, DMT). DMA is direct memory access where the starting address and word count are kept in the register set. Up to eight DMA channels (total) are supported by the computer. DMC is very similar to DMA, and stands for direct memory channel. In this case the starting and ending addresses are stored in high speed memory, providing up to 2000 DMC channels. DMT is direct memory transfer, where the data address and word count are maintained external to the computer. The address must be applied to the bus with the data when requesting a DMT. This allows a random accessing of memory. In our case, DMT is preferred because the laser data returns will not be in any particular order. This is because the mast controller fires the laser as soon as the angular position of the mirror corresponds to one of the desired elevation angles.

Initially, rather than wasting the time building and debugging the full-blown interface, it was decided to build a simple test interface. This would allow testing of both hardware and software concepts. The interface would test PIO, a hardwired vectored interrupt, and a hardwired DMT. Initially, nothing worked. After talking with Prime for a while, it was found that our GPIB documentation was incomplete, and in a few minor locations, incorrect. Eventually, after more testing, the board started to show signs of life. The PIO, interrupt and DMT all worked fine separately, but when a vectored interrupt was alternated with a DMT, strange results occurred. The system would run for about 20 seconds and then crash. Using a program to display the data as it came in, bad data words could be seen every once in a while. As another symptom, whenever the disk was active (reading or writing), the system would crash even faster. This disk-GPIB interaction seemed to imply some kind of priority problem, but little difference was observed as the GPIB priority was changed through every level. The same result occurred when our GPIB was tested on a different Prime 500, thus eliminating a hardware problem on our particular computer. It is still not known whether the problem is in hardware or software and even Prime is unable to supply an explanation.

It was decided to put our upper level realtime interface aside and work on a degraded interface which would allow the input and storage of data for off-line processing. Once data is available for software testing, the realtime interface testing can resume.

For a description of the upper level interface see Figure 9. Each telemetry data word received by the interface is composed of 16 bits of information and 16 bits of address or identifier. The address also contains any mast interrupts such as end of scan or end of azimuth. This 32 bit word is received in serial and converted to parallel. In the upper level interface, a portion of the address will be concatenated with a register containing an offset address into real memory. DMT will be used to put the data into memory. Finally there will be one vectored interrupt which uses a status register to identify various conditions such as mast interrupts, data overruns, or timeouts.

The temporary degraded interface can be seen in Figure 10. DMC will be used to insert all 32 bits of the telemetry data word, and DMC will be followed by an interrupt. It will then be up to the interrupt service routine to decide where to put the data from the address and to strip the scan status from the address part of the data. The

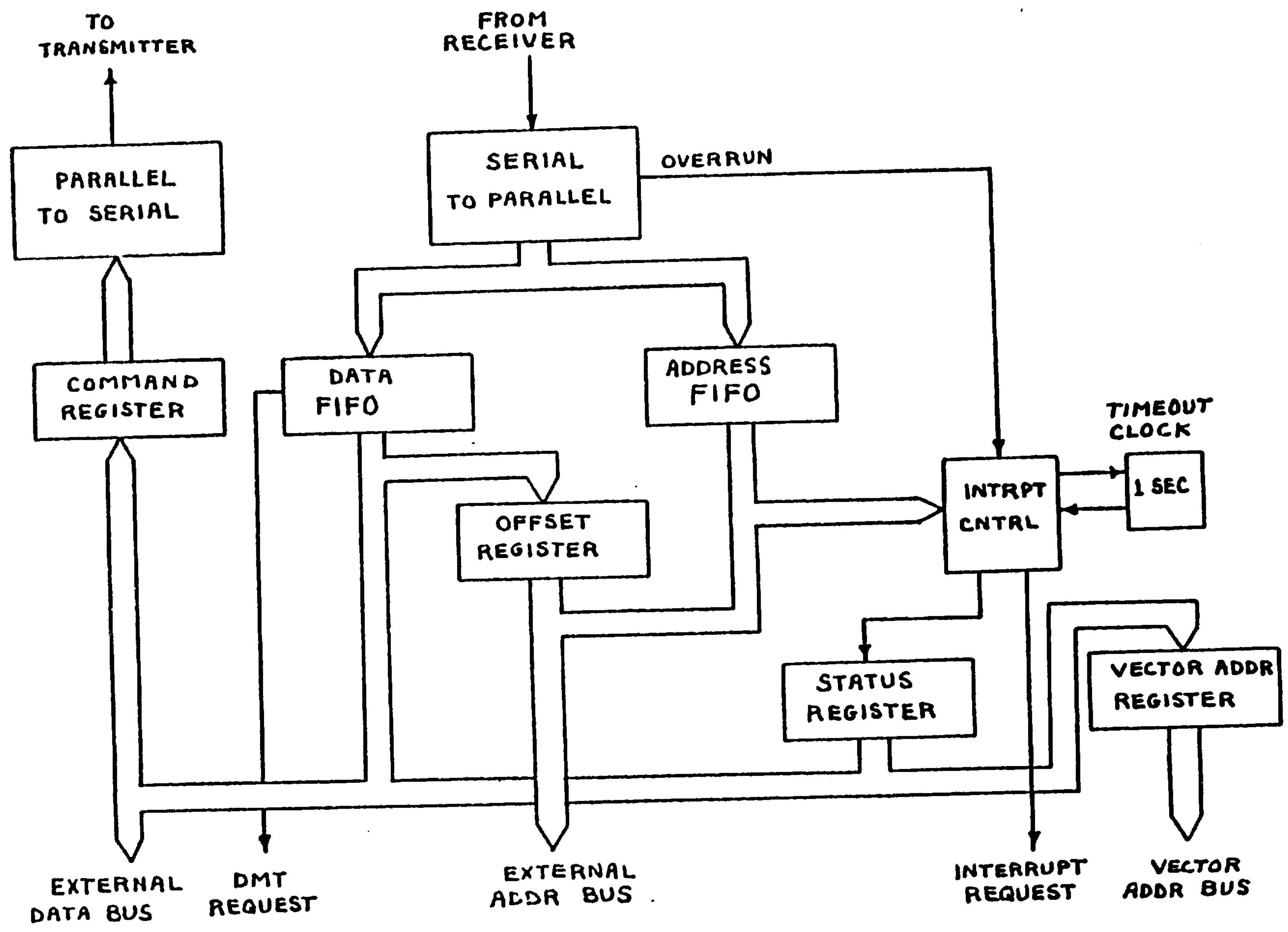


FIGURE 9. Realtime Interface

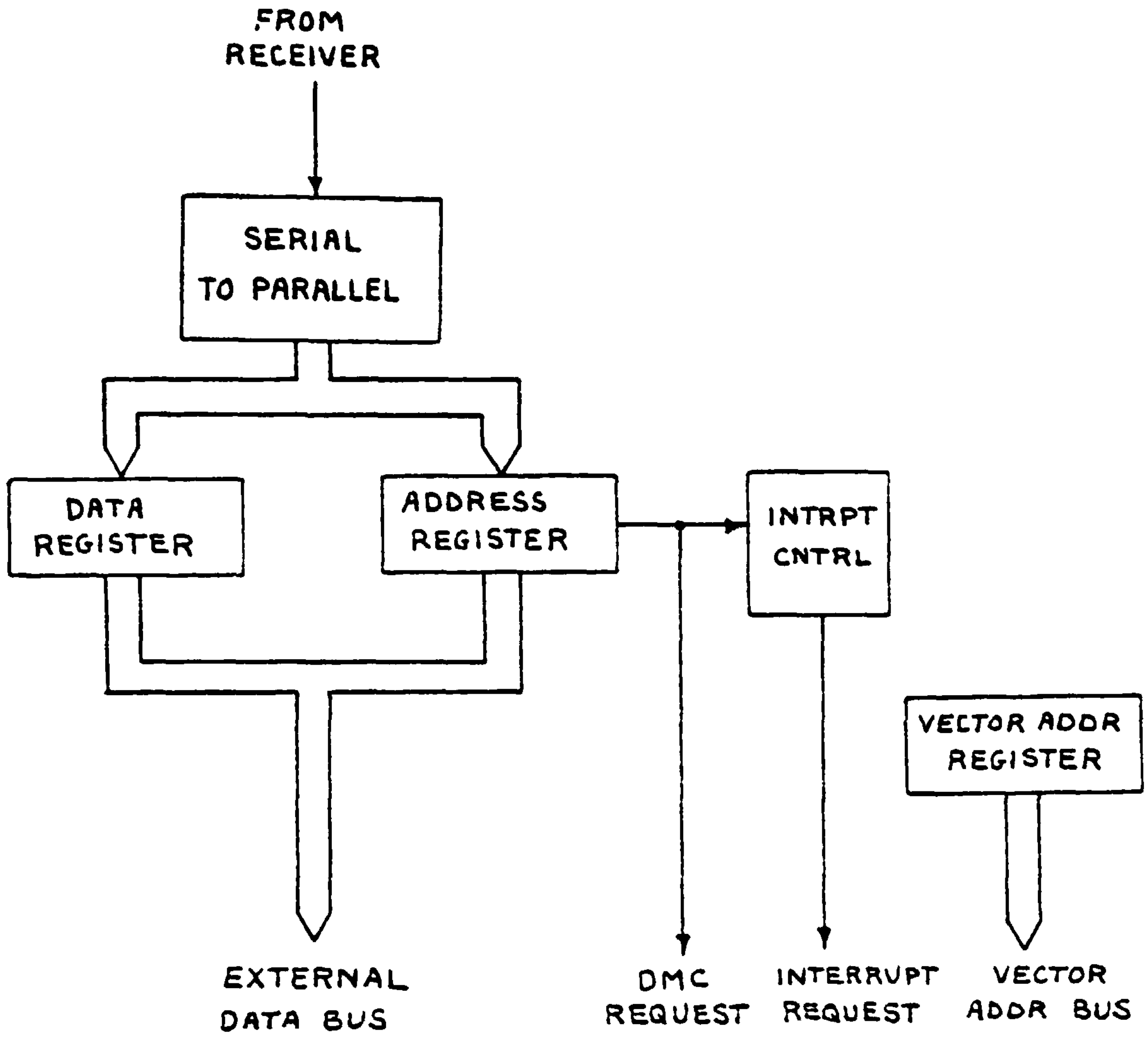


FIGURE 10. Temporary Interface

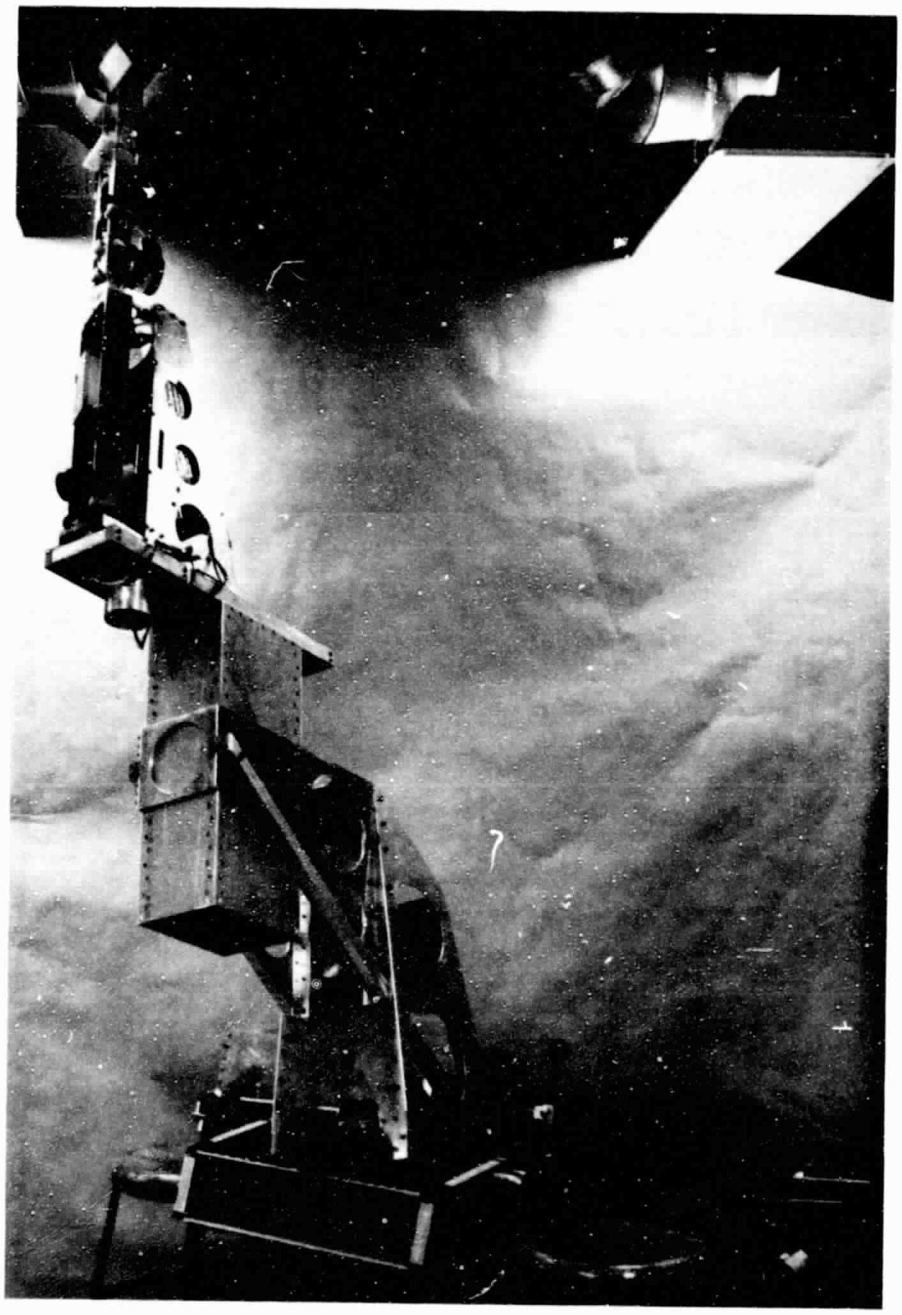
degraded interface will not allow the realtime processing of data, but should enable the accumulation of up to about 20 scans of data for off-line post processing.

D. The Dynamic Test Platform

A dynamic test platform on which the elevation scanning mast can be mounted is currently under construction (see Figure 11). The idea is to enable accurate in-house testing of the hardware and software responsible for hazard detection.

The platform is motor driven such that the pitch and roll of the mast is dynamically variable. Both pitch and roll are separately controlled and each has variable amplitude and rate. An attitude gyro is mounted on the platform to provide pitch and roll data to the computer.

This platform is expected to aid a great deal in software testing since the actual laboratory scene can be accurately compared to the computer results. It should also help in the difficult job of calibrating the optics.



MAST MOUNTED ON THE DYNAMIC TEST PLATFORM

Figure 11

REALTIME SUPPORT SOFTWARE

A. Software Overview

The new realtime software has been written on the Prime 500 minicomputer and is intended for the higher level interface. Because of unsolved problems with the interface it may become necessary to modify the software; however, the basic flow can be left intact.

The objective of the realtime software is still to implement control of the new Mars rover and to record the raw data from either the rover or the platform (see Figure 12). Most of the software is written in FORTRAN; though, some assembler was used.

The basic data consists of an interrupt status flag, laser data, and vehicle state data (see Figure 13). The possible interrupt status flags include:

EOA: End of azimuth; the data consists of laser returns and vehicle state information.

EOS: End of scan; same data as EOA, but also signals that a full scan has been taken.

VI: Vehicle interrupt; the data consists of vehicle state information only.

Timeout: No interrupts have been received for at least one second; it signals a possible hardware problem.

Overrun: New data has written over old data before old data was read; stop vehicle and wait for next EOS before accepting new data.

Telemetry data will enter the Prime via DMT into an azimuth buffer, one azimuth at a time, followed by the appropriate interrupt. Azimuths can be broken into two types: a laser azimuth and a vehicle

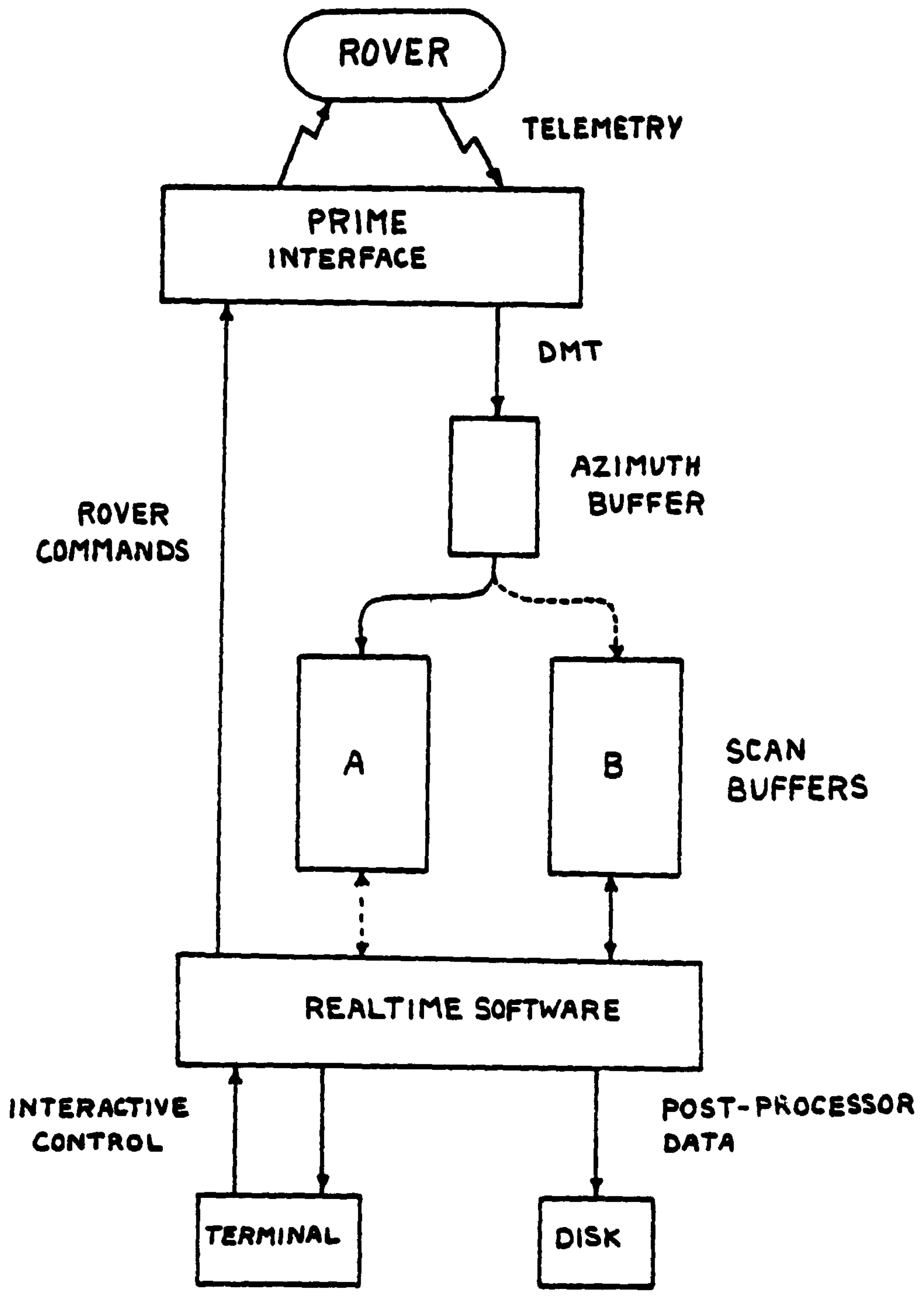


FIGURE 12. Realtime Hardware - Software Interface

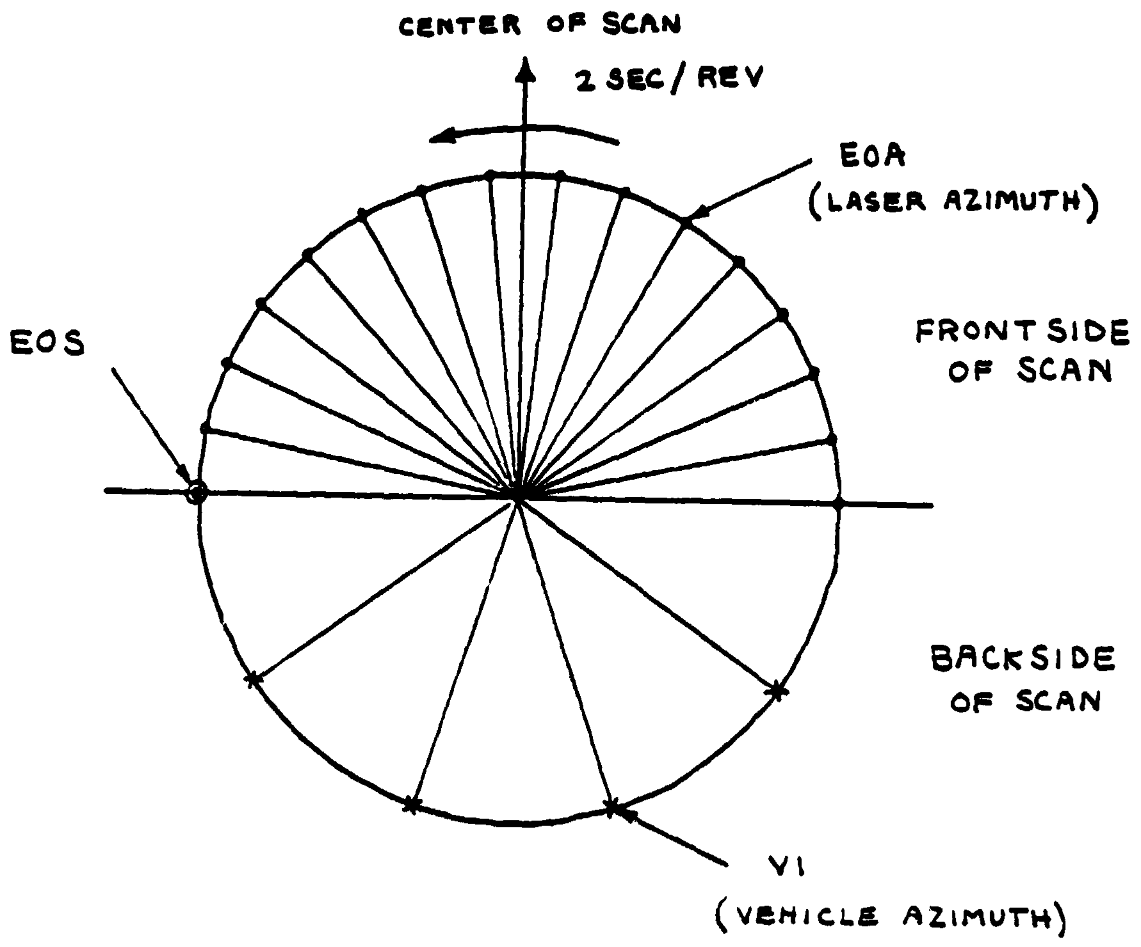


FIGURE 13. Scan Timing

azimuth. A laser azimuth would occur on the frontside of a scan and would be followed by an EOA or EOS. A vehicle azimuth would occur on the backside of a scan and would be followed by a VI. Because the vehicle data is not needed as often on the backside of a scan (for navigation), the vehicle azimuths can occur less frequently.

From the azimuth buffer, the data is moved to one of two scan buffers. Each scan buffer is large enough to hold an entire scan with the azimuths stored sequentially. There are two of them to provide a double buffering scheme such that the software can be processing one scan while another is arriving.

A macro description of the overall realtime system will now be given (see Figure 14). The main routine, called EXEC, is in charge of the entire system flow. After the user gives the RUN command, the system is initialized. The system then waits for an interrupt to signal that some data is available. After an interrupt, NAVIG is called to convert the data to a usable format and to perform navigation. If an EOA interrupt occurred, then the data includes laser returns, and the MODEL routine is called to analyze these returns.

The terrain modeller analysis can be further broken down into inpath and crosspath. Inpath is along an azimuth and can be done for each azimuth as it arrives. Crosspath is along the scan and can only be done after the EOS interrupt occurs. If an EOS did occur, then after performing a crosspath analysis, the modeller would pass its results to the path selection routine (PSA). Using both current and past information, the PSA would select an optimal path and send the appropriate turn command to the rover. Finally the data would be saved for

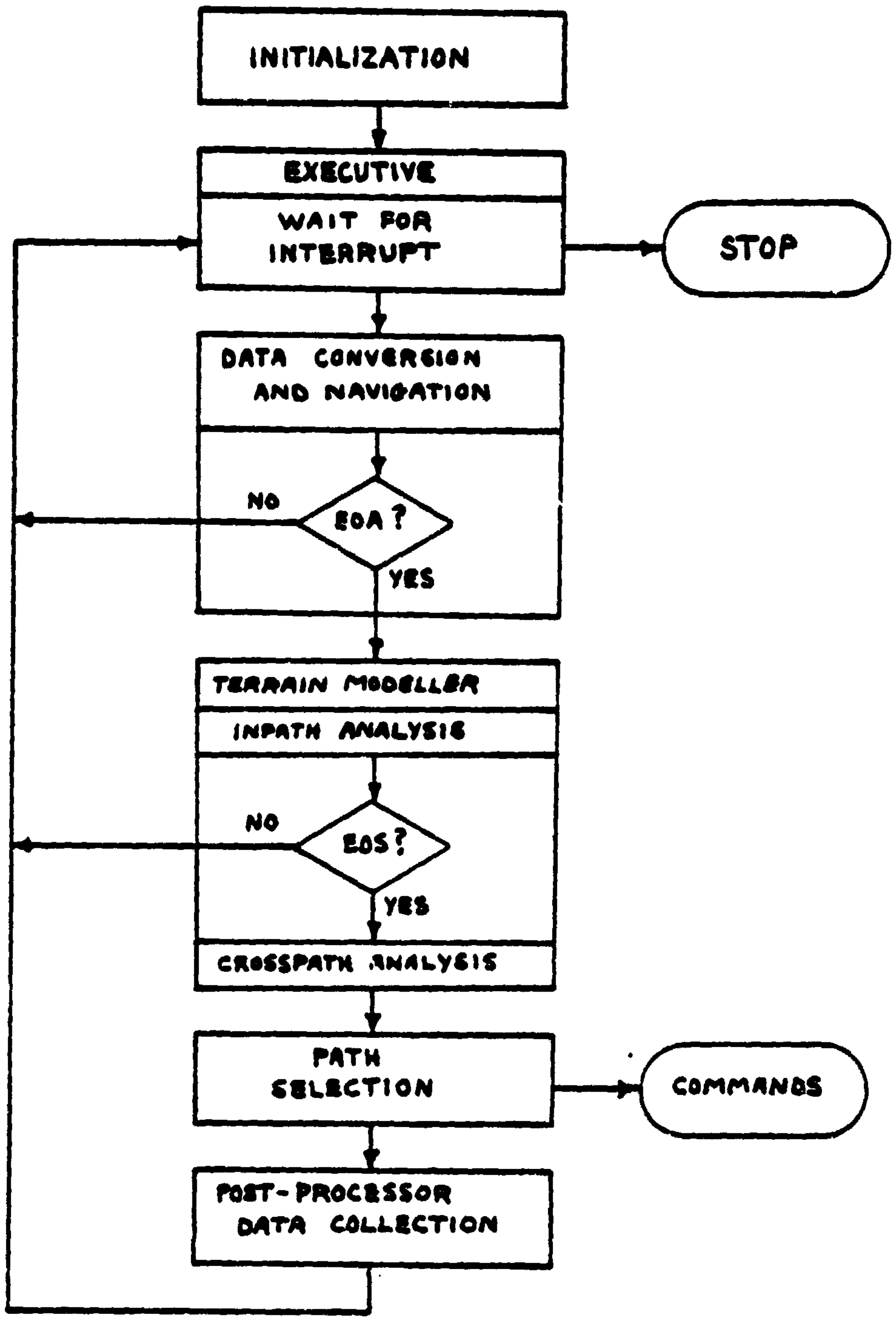


FIGURE 14. Realtime System Flow

the post-run analysis and the process would be repeated.

B. Lower Level Routines

Many modifications had to be made to the Prime operating system to implement realtime control of the rover. This was further complicated by the fact that the operating system was more suited to timesharing rather than to realtime control. The Prime operating system (PRIMOS) is very complex and therefore will not be described here. More information can be found in the Prime manuals and listing (see Reference 7).

PRIMOS is a multilevel operating system existing in levels II, III, IV, and V. Our operating system is a modified version of PRIMOS V Rev 15.0.* Note that moving to a new revision may require significant changes. Space was left in the operating system for the addition of new system processes by the addition of two spare templates: SP1 and SP2. We took over the SP2 template throughout. Besides those changes, three routines encompass the major additions to the operating system. These are DEVEIO, MRVDIM and T$ROVR. Also a new common block, MRVCOM, was added. Flowcharts for all of the realtime support routines can be found in Appendix. B.

B.1. DEVEIO

DEVEIO is a two argument system subroutine, written in assembler language, which allows FORTRAN programs to execute I/O instructions. The first argument is the instruction, function, and device code while the second argument is an input or output if required. It is called as a FORTRAN function subroutine and returns true if success-

*Note that at the time of this writing, our version of the operating system is being updated to Rev. 16.0.

ful and false if unsuccessful.

B.2. MRVDIM

Vectored interrupts from the GPIB interface go through location 161_8. The interrupt service routine will automatically disable any further interface interrupts. It will then NOTIFY the interrupt process MRVDIM by the use of the MRVSEM semaphore and return.

A semaphore is a two word software device used to asynchronously start-up or schedule other processes. The first word is the semaphore count, and the second word is a pointer to the wait list. There are two operations that can be performed on a semaphore by a user. A user can NOTIFY a particular semaphore, which just decrements that semaphore count. When the computer gets around to examining the semaphores, if it finds any that are less than or equal to zero, it takes the highest priority process from that wait list and puts it on the ready list. The highest priority process on the ready list runs. A user can also WAIT on a semaphore, which increments the count, and puts that process on the wait list. Note that if a process does a WAIT, and the count remains less than or equal to zero, that process stays on the ready list.

Initially the MRVSEM semaphore starts out with a count of one, and MRVDIM on the wait list. When the MRVDIM process is started, it disables any further DMX and inputs the interrupt status register. It then checks the status for a hardware overrun condition. This could occur if the MRVDIM process took too long with the last set of azimuth data, therefore losing some of the next azimuth data. On an OVERRUN, the stopped flag (RVSTOP) is checked. RVSTOP declares whether the rover is currently halted by the MRVDIM process for either a hardware or software overrun. A software overrun occurs when the user process is not done with either scan buffer before MRVDIM receives more data.

If RVSTOP is set then ignore the OVERRUN since the vehicle is already stopped. If RVSTOP is not set, then put a -2 in the current buffer status word to signal an OVERRUN and NOTIFY the MRVFUL semaphore to start up the user process. Also, set RVSTOP and send a HALT command to the vehicle. MRVDIM will now wait until the next EOS so it can re-synchronize the scan. Finally, whether RVSTOP was set or not, re-enable DMX and interrupts and WAIT on MRVSEM.

If there was no OVERRUN then check the status for a TIMEOUT condition. This interface generated interrupt signals a lack of external interrupts for at least one second. On a TIMEOUT, put a -3 into the current buffer status word to signal TIMEOUT and again NOTIFY the MRVFUL semaphore. Then send a HALT command to the rover and WAIT on MRVSEM. Note that by not re-enabling interrupts the MRVDIM process cannot be restarted.

When there is no OVERRUN or TIMEOUT, examine RVSTOP. If RVSTOP is set, then check the status for an EOS. If no EOS then ignore the interrupt, re-enable DMX and interrupts and WAIT on MRVSEM. But if an EOS was received then try to re-synch the scan. Check to see if one of the scan buffers has been declared empty by the user process. If there is no empty buffer, then do nothing except re-enable DMX and interrupts, and WAIT on MRVSEM for another EOS. If there is a free buffer, then restart the rover. Then set up the new buffer pointers and clear RVSTOP. Finally, re-enable DMX and interrupts, and WAIT on

MRVSEM.

If there was no OVERRUN or TIMEOUT, and RVSTOP was not set, then there is some azimuth data to move. First get the current time (TIMNOW and VCLOK), accurate to 1/330 second, and store it in the last two words of the azimutn buffer. Then move the azimuth buffer to the current scan buffer. Since both buffers should be locked in memory (from being paged out), if a fault occurs on the data move instruction (ZMVD), the user process must have abnormally terminated execution. In this case send a HALT command to the rover and again WAIT on MRVSEM without re-enabling interrupts. This is the system failsafe; if the user process terminates, the rover is automatically halted.

If there was no fault, then the move was successful. In that case NOTIFY MRVFUL, update the pointer to the scan buffer and then check for an EOS. If no EOS then again re-enable DMX and interrupts and WAIT on MRVSEM. But if an EOS was received, then put the positive azimuth count in the buffer status word to signal an EOS to the user. If the other scan buffer is empty then set up the pointers to load into it next. If it is not empty, then set RVSTOP and halt the rover. Finally, empty or not, re-enable DMX and interrupts and WAIT on MRVSEM.

B.3. T$ROVR

T$ROVR is a system routine written to simplify and protect the user-system software interface. It is a five argument FORTRAN subroutine, and has five basic functions. It allows the user to initialize the interface, to stop the interface, to send rover commands, to empty a scan buffer, and to WAIT on MRVFUL. The first argument selects the function. Error-checking with appropriate messages safeguards against incorrect usage.

T$ROVR first checks to see if the rover is assigned to the user. The rover has been made an assignable device, and as such must be assigned and unassigned using:

ASSIGN ROVER
UNASSIGN ROVER

Next it makes sure that the first call to T$ROVR is an initialization instruction. Then it jumps to the function selected by the first argument.

Initialization is the most complicated function. First, the scan buffer addresses, arguments 4 and 5 are checked for valid addresses. Then the azimuth buffer size and number, arguments 2 and 3, are checked for validity. The azimuth buffer size is restricted to be between 1 and 1022, while the azimuth number must be between 1 and 256. Now T$ROVR checks to see if this is the first initialization call. If so then the device status is checked using DEVEIO to see if the GPIB replies. If everything is satisfactory then MAPIO is called to map the 1024 word DMX buffer into the MRVDIM azimuth buffer. Then LOCKPG is called to lock MRVDIM, the azimuth buffer and the two scan buffers into memory. Finally, using DEVEIO, DMX and interrupts are enabled and an INIT command is sent to the rover. The MRVFUL semaphore count is zeroed and T$ROVR returns.

The stop function starts by looping until the RVSTOP flag is set signalling that the rover has been stopped. It then calls UMAPIO to unlock or free the two scan buffers. Using DEVEIO, it again sends the HALT command to the rover, disables DMX and interrupts and then

T$ROVR returns.

The command function calls DEVEIO to send a command to the rover. It also loops until it is successful and then returns.

The empty function is used to declare a scan buffer empty and to switch to the other buffer. The second argument selects the buffer being emptied. Note that successive calls must alternate buffers. Since an OVERRUN may have stopped the rover, the third argument may be used to specify what restart command, if any, should be given. RVFILL is a variable used to keep track of which buffer is currently being filled, and RVNEXT, the buffer to fill next. The MRVDIM process will fill the buffer selected by RVFILL, move RVNEXT into RVFILL, and clear RVNEXT. If MRVDIM finds RVFILL equal to zero, then no buffers are empty and a software OVERRUN occurs. The empty function sets up RVFILL and RVNEXT.

Last is the WAIT function. It checks RVFILL to make sure at least one scan buffer is empty. It then does a WAIT on the MRVFUL semaphore.

C. System Routines

The system routines make up the user process and include the realtime system executive and its subroutines. Two of these subroutines: the terrain modeller (MODEL) and the path selection routine (PSA) will not be discussed here.

C.1. EXEC

The main upper level routine is the Mars system executive. This is the user process which performs the realtime analysis and control. EXEC basically controls the flow of the system, with the real work done by subroutines (see Figure 15). All of the routines are linked together by the main common block RVRCOM (see Table 1). Once EXEC is started, the user has the ability to run as many realtime experiments as desired without restarting the system. The user can change parameters, select different PSA and MODEL routines, and even run the rover manually from the keyboard.

On starting the system, EXEC first tries to initialize the important runtime parameters with default values. To do this it ATTACHes to the MARS.DATA subUFD (sub-User File Directory) and makes that its home UFD. This is done to keep all the runtime data files separate from any programs being developed, and to try to keep them all together. EXEC then opens the MARS.DEFAULT file and reads the default values. Keeping default values in a separate file rather than in the EXEC program itself makes them easier to change and does not require a recompilation of EXEC. The parameters initialized and the file format is shown in an example copy of the MARS.DEFAULT file (see Table 2). Note also that new parameters can be added easily.

EXEC displays the parameter values and then enters the interaction processor. The interaction processor is just a segment of code which allows the user to enter commands. The prompt for the interaction processor is "CO:", and the command line syntax is:

```
<COMMAND><= or , or (blank)><VALUE>
```

Note that for some commands, the delimiter and value may not be required. The value may be either real or integer, and will be converted automatically if necessary. Also, if a required value is omitted, the process will ask for it. The possible commands are kept in the array

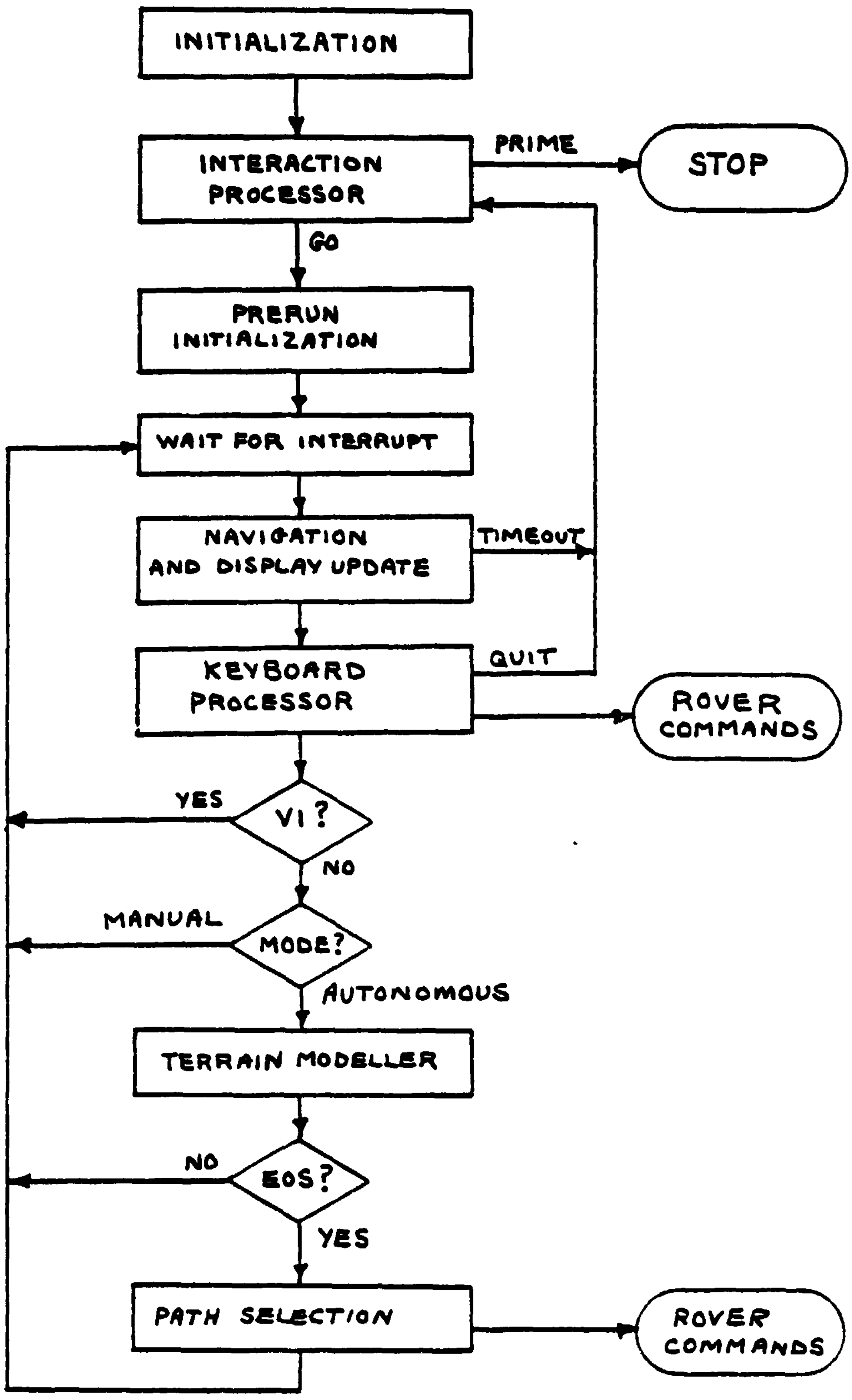


FIGURE 15. Flow Diagram of EXEC

```
COMMON /RVRCOM/       /* Realtime System Common
      BUFA(2048)      /* Buffer for Raw Data
      BUFB(2048)      /* Buffer for Raw Data
      LBUFF(1024)     /* Buffer of Lower Laser Data
      UBUFF(1024)     /* Buffer of Upper Laser Data
      PITCH(64)       /* Vehicle Pitch per Azimuth (Rads)
      ROLL(64)        /* Vehicle Roll per Azimuth (Rads)
      HEADNG(64)      /* Vehicle Heading per Azimuth (Rads)
      AXROLL(64)      /* Front Axle Roll per Azimuth (Rads)
      STEER(64)       /* Steering Angle per Azimuth (Rads)
      SPEED(64)       /* Vehicle Speed per Azimuth (M/Sec)
      DTIME(64)       /* Delta Time between Azimuths (Sec)
      XLOC(64)        /* X Location of Vehicle (M)
      YLOC(64)        /* Y Location of Vehicle (M)
      ZLOC(64)        /* Z Location of Vehicle (M)
      TIME            /* Total Time Since Beginning of Run (Sec)
      DHEAD           /* Desired Heading (Rad)
      NUMAZL          /* Number of Laser Azimuths per Scan
      NUMAZV          /* Number of Vehicle Azimuths per Scan
      NUMLAZ          /* Number of Laser Shots per Azimuth
      NUMSEN          /* Number of Sensors per Azimuth
```

TABLE 1a. RVRCOM

XFINAL	/* X Location of Target (M)
YFINAL	/* Y Location of Target (M)
DELTXY	/* Desired "Closeness" to Target (M)
IMOD	/* Version of MODELLER to Use
IPSA	/* Version of PSA to Use
DSPTIM	/* Time between Display Updates (Sec)
NUMAZT	/* Total Number of Azimuths per Scan
SCAN	/* Current Scan Number
AZMUTH	/* Current Azimuth Number
MAPX	/* Map X Axis Range (M)
MAPY	/* Map Y Axis Range (M)
ORIGX	/* Map X Origin
ORIGY	/* Map Y Origin
MRVCMD	/* Command Sent to Rover
OVRRUN	/* Buffer Overrun Flag
TIMOUT	/* Timeout Flag
EOA	/* End of Azimuth Flag
VI	/* Vehicle Interrupt Flag
EOS	/* End of Scan Flag
INIT	/* Initialization Flag
RUN	/* Run Flag

TABLE 1b. RVRCOM

* MARS.DEFAULT File: A Sample

16	NUMAZL: Number of Laser Azimuths per Scan
4	NUMAZV: Number of Vehicle Azimuths per Scan
20	NUMLAZ: Number of Laser Shots per Azimuth
20	NUMSEN: Number of Sensors per Azimuth
10.0	XFINAL: Target X Coordinate
10.0	YFINAL: Target Y Coordinate
0.5	DELTXY: Desired "Closeness" to Target
1	IMOD: MODELLER Version
1	IPSA: PSA Version
20.0	MAPX: Map X Axis Range
20.0	MAPY: Map Y Axis Range
0	ORIGX: Map X Origin
0	ORIGY: Map Y Origin
0.25	DSPTIM: Time between Display Updates

TABLE 2. MARS.DEFAULT File

TBL1, and provide for easy expansion (see Table 3).

The interaction processor starts by inputting up to 40 characters from the keyboard into the array KBUF. A subroutine, called GETTOK, strips the first token from the command line, which should be the command, and returns it in the array IBUF. Another subroutine, GETCMD, then searches the TBL1 array for a match to IBUF. If there is no match then the message

ILLEGAL COMMAND

is output and the processor goes back for another command. If a match is found then the command number is returned in the variable CMD. Each command in TBL1 is followed by a key which tells what kind of value, if any, should follow the command. CMD is used to index into TBL1 to get this key, which is assigned to the variable KEY. If KEY equals zero then there are no parameters and so EXEC jumps to execute that command through a computed GOTO, indexed by CMD. Otherwise GETTOK is called again to get the parameter. If the parameter is missing, it is requested only once using the prompt: "PAR=". And if the user enters nothing, then the command is ignored. If there is a parameter then the subroutine CNVPAR is called to convert the parameter from ASCII to either integer or real as determined by KEY. If there is an error on the parameter conversion then the message

ILLEGAL PARAMETER
PAR=

is issued and the user can try again. Finally, EXEC jumps through a computed GOTO, indexed on CMD, to execute the command. Note that when executing a command with a parameter, the parameter is always checked

DISPLY		Display default parameter values.
PRIME		Return to PRIMOS.
NUMAZL	⟨Value⟩	Change NUMAZL to ⟨Value⟩.
NUMAZV	⟨Value⟩	Change NUMAZV to ⟨Value⟩.
NUMLAZ	⟨Value⟩	Change NUMLAZ to ⟨Value⟩.
NUMSEN	⟨Value⟩	Change NUMSEN to ⟨Value⟩.
XFINAL	⟨Value⟩	Change XFINAL to ⟨Value⟩.
YFINAL	⟨Value⟩	Change YFINAL to ⟨Value⟩.
DELTXY	⟨Value⟩	Change DELTXY to ⟨Value⟩.
IMOD	⟨Value⟩	Change IMOD to ⟨Value⟩.
IPSA	⟨Value⟩	Change IPSA to ⟨Value⟩.
MAPX	⟨Value⟩	Change MAPX to ⟨Value⟩.
MAPY	⟨Value⟩	Change MAPY to ⟨Value⟩.
ORIGX	⟨Value⟩	Change ORIGX to ⟨Value⟩.
ORIGY	⟨Value⟩	Change ORIGY to ⟨Value⟩.
DSPTIM	⟨Value⟩	Change DSPTIM to ⟨Value⟩.
GO		Begin accepting realtime data.

TABLE 3. Interaction Processor Commands

for validity first.

On a GO command, all necessary flags and counters are initialized first. To enable the PSA and MODEL routines to initialize themselves (since different versions may exist), they are called with the variable INIT set true. Next, T$ROVR is called to initialize the interface. A post-processor file is then opened to hold the runtime data for this particular run. Each realtime experiment gets a new post-processor file with a unique name of the form

M. MM / DD / YY .# NN

where MM, DD, and YY are the two digit representations of the current month, day, and year respectively. NN is a two digit number between 00 and 99, starting at 00 and incrementing for each new run of that particular day. After opening this file, EXEC writes an information record with the format shown in Table 4.

Next, the CRT screen is set up for the runtime display (see Figure 16). During the run, these parameters will be updated periodically. The STATUS variable will show the current interrupt type: EOA, EOS, VI, TIMEOUT, or OVERRUN. The COMMAND label will show the last command sent to the rover, and the TTY CO label will show the operator command input during runtime.

The pre-run initialization concludes by calls to T$ROVR to empty both scan buffers and a call to NAVIG to allow it to initialize itself. Finally, EXEC does a WAIT call using T$ROVR which begins the realtime execution phase of EXEC.

On an interrupt, the buffer result subroutine, BUFRES, is called. BUFRES checks the reason for the interrupt, stores the raw

WORD	CONTENTS
1	Record Length (=20 words).
2	Record Identifier (-1 = Information Record).
3	Current Month (01 - 12).
4	Current Day (01 - 31).
5	Current Year (00 - 99).
6	Current Time (24:00).
7	NUMAZL: number of laser azimuths.
8	NUMAZV: number of vehicle azimuths.
9	NUMLAZ: number of lasers per azimuth.
10	NUMSEN: number of sensors per azimuth.
11-12	XFINAL: target X coordinate.
13-14	YFINAL: target Y coordinate.
15-16	DELTXY: desired "closeness" to target.
17	IMOD: version of MODELLER used.
18	IPSA: version of PSA used.
19-20	DSPTIM: time between display updates.

TABLE 4. Post-Processor Information Record

```
***********************************************     TIME:
*                                             *     HEADING:
*                                             *     PITCH:
*                                             *     ROLL:
*                                             *     X LOC:
*                                             *     Y LOC:
*                                             *     AX ROLL:
*                                             *     SPEED:
*                                             *     STEER:
*                                             *     LASER:
*                      R                      *     STATUS:
*                                             *     COMMAND:
*                                             *
*                                             *     TTY CO:
*                                             *
*                                             *
*                                             *
*                                             *
*                                             *
***********************************************

X RANGE:                Y RANGE:
X ORIG:                 Y ORIG:
```

FIGURE 16. Runtime Display

data, calls NAVIG, and updates the display by calling the subroutine SCREEN. If a TIMEOUT interrupt was received then the run is halted and control returns to the interaction processor. Otherwise a segment of code called the keyboard processor is entered.

The keyboard processor allows the operator to enter commands during runtime. At runtime, the terminal is automatically placed into half-duplex mode, so that nothing the operator types will show up on the screen. This is necessary since the cursor is flying all over the screen updating the map and the display parameters. The keyboard processor reads what is typed on the keyboard from an internal buffer, and displays it following the TTY CO label.

A subroutine called KEYBIN reads and displays the keyboard input. When it comes across a line-feed character, it sets the flag ENTER true, and returns the text in the array BUF. The keyboard processor then uses the subroutines GETTOK, GETCMD and CNVPAR just as in the interaction processor. A different command table, CMDTBL, is used to contain the possible runtime commands (see Table 5). Also because of limited space, an error is signalled by an asterisk as

* TTY CO:

An At character "@" has the effect of erasing the current command.

Note that after a GO command in the interaction processor, data will be accepted, stored, and decoded, but the rover will be in the manual mode. Therefore, the PSA and MODEL routines will be skipped until the RUN command is given.

Finally, if EXEC is in the autonomous control mode, the selected MODEL routine will be called. Then, if an EOS interrupt was re-

RUN	Begin autonomous control.
STOP	Stop rover.
F ⟨Value⟩	Forward command where:
	F 1 Slow
	F 2 Medium
	F 3 Fast
R ⟨Value⟩	Reverse command where:
	R 1 Slow
	R 2 Medium
	R 3 Fast
T ⟨Value⟩	Turn command where: ⟨Value⟩ can take on the values (-7 to +7) and refers to 1 of 15 absolute steering angles.
	T -7 90° Left turn
	T 0 0° Turn
	T 7 90° Right turn

TABLE 5. Keyboard Processor Commands

ceived, the selected PSA routine will be called. The PSA will send commands to the rover through a T$ROVR call and then the used scan buffer will be emptied by another T$ROVR call. Whether an EOS was received or not, the azimuth and scan counts (AZMUTH, SCAN) are updated and EXEC jumps back to do a T$ROVR WAIT call on MRVFUL again.

C.2. NAVIG

The NAVIG subroutine has the job of converting the raw data to a form which can be used by the other programs, as well as keeping track of navigation. The raw azimuth data format is shown in Table 6. The individual word formats can be found in Appendix A.

NAVIG is a four argument subroutine called from the BUFRES subroutine.

NAVIG (VBUF, LAZBUF, UPBUF, LOBUF)

The argument VBUF is the address of the start of the current vehicle data buffer located within the scan buffer. LAZBUF is the address of the start of the current laser data buffer, also located within the current starting addresses in the UBUFF and LBUFF arrays respectively. Those arrays hold the upper and lower receiver cone numbers for each laser shot.

In the beginning of NAVIG, the routine checks the variable INIT, located in RVRCOM, to see if it should initialize itself. There are many variables to zero out since NAVIG uses a cumulative technique to perform navigation. If INIT is false, then NAVIG begins to decode the data.

The time, in seconds, since the beginning of the run is computed and kept in the variable TIME. Note that the MODEL routine uses

WORD	CONTENTS
1	Not used.
2	Raw Steering Angle.
3	Raw Left-Rear Tachometer Reading.
4	Raw Right-Rear Tachometer Reading.
5	Not used.
6	Not used.
7	Not used.
8	Not used.
9	Raw Front Axle Roll Angle.
10	Raw Roll Angle.
11	Raw Pitch Angle.
12	Not used.
13	Raw Right-Front Tachometer Reading.
14	Not used.
15	Raw Left-Front Tachometer Reading.
16	Raw Heading Angle.
17	Raw Time (minutes).
18	Raw Time (330's of second).

TABLE 6. Azimuth Buffer Format

a lot of information on a per azimuth basis when it does a crosspath analysis. Therefore, an array, DTIME, is formed to contain the time between azimuths for each azimuth.

Next the heading angle, in radians, is computed and stored in the array HEADNG, one entry per azimuth. It is necessary to touch on the matter of bad data. Since data transmitted from the vehicle might be lost because of interference, it is important to detect this error and to minimize its damage. The solution is to make the interrupt process, MRVDIM, pack the empty azimuth buffer with impossible data. Luckily, an all ones pattern cannot appear in the current data chosen. To minimize the damage, NAVIG will substitute the last valid value of that particular variable. It will also put an asterisk next to that label on the display screen. If the operator sees that a data word is consistantly wrong, then he could stop the rover, since there might be a hardware problem. In the laser data, a missing return will be substituted for a bad data word and the number of bad laser data words will be displayed next to the LASER label on the screen.

After the decoded heading angle has been saved, the pitch, roll and front-axle roll are decoded and stored in the arrays PITCH, ROLL, and AXROLL respectively. The speed is calculated from the average of the two front wheel tachometer readings, and stored in the array SPEED. Next, the X, Y, and Z locations of the rover are calculated using the formulas given in Table 7. They are stored in the arrays XLOC, YLOC, and ZLOC respectively.

Now NAVIG checks to see if it is processing a VI interrupt. If so, then it returns because there is no laser data. If it is pro-

$$\text{AVG(SPEED)} = \frac{\text{SPEED(K) + SPEED(K-1)}}{2.0}$$

$$\text{BETA(K)} = \text{HEADNG(K) + STEER(K)}$$

$$\text{AVG(BETA)} = \frac{\text{BETA(K) + BETA(K-1)}}{2.0}$$

$$\text{AVG(PITCH)} = \frac{\text{PITCH(K) + PITCH(K-1)}}{2.0}$$

$$\text{XLOC(K)} = \text{XLOC(K-1) + AVG(SPEED)COS(AVG(BETA))}$$

$$\text{YLOC(K)} = \text{YLOC(K-1) + AVG(SPEED)SIN(AVG(BETA))}$$

$$\text{ZLOC(K)} = \text{ZLOC(K-1) + AVG(SPEED)SIN(AVG(PITCH))}$$

where VARIABLE(K) = Current value

VARIABLE(K-1) = Last value

TABLE 7. Navigation Formulas

cessing an EOA or EOS, then there is laser data. The upper and lower receiver cone numbers must be separated from each laser data word and stored in the arrays UPBUF and LOBUF respectively. Data errors are made into missing returns, and a count is kept in the variable LDERR. Finally NAVIG returns.

C.3. BUFRES

BUFRES is a subroutine which determines the interrupt status, saves the raw data, calls NAVIG to perform conversions and navigation, and calls SCREEN to update the display. It has one argument, BUF,

```
BUFRES (BUF)
```

which is the current scan buffer address.

BUFRES first decodes the interrupt status from the first word of the current scan buffer and outputs it to the display next to the STATUS label. If a TIMEOUT or OVERRUN occurred, then that information is written to the post-processor file and BUFRES returns. On an EOA, EOS, or VI interrupt, a similar post-processor record is written with the format given in Table 8. Next, the starting addresses for the vehicle data and laser data within BUF, and the laser data destination within UBUFF and LBUFF are calculated, and used in the call to NAVIG. Finally the subroutine SCREEN is called to update the CRT display.

C.4. GETTOK

GETTOK is a general purpose subroutine which will parse a command line, returning tokens on consecutive calls. The delimiters recognized include a blank, a comma, and an equals sign. The Prime subroutine RDTK$$ also returns tokens but it uses a different set of

WORD	CONTENTS
1	Record Length.
2	Record Identifier where: -1 = Information Record 1 = EOA Record 2 = VI Record 3 = EOS Record 4 = OVERRUN Record 5 = TIMEOUT Record 6 = MODELLER Record 7 = PSA Record
3	Last command sent to rover.
4	Current Scan Number.
5	Current Azimuth Number.
6	Not used.
7	Raw Steering Angle.
8	Raw Left-Rear Tachometer Reading.
9	Raw Right-Rear Tachometer Reading.
10	Not used.
11	Not used.

TABLE 8a. Post-Processor General Record

WORD	CONTENTS
12	Not used.
13	Not used.
14	Raw Front Axle Roll Angle.
15	Raw Roll Angle.
16	Raw Pitch Angle.
17	Not used.
18	Raw Right-Front Tachometer Reading.
19	Not used.
20	Raw Left-Front Tachometer Reading.
21	Raw Heading Angle.
22	Raw Time (minutes).
23	Raw Time (330's of second).
24-up	Raw Laser Data (if EOA or EOS).

TABLE 8b. Post-Processor General Record

delimiters.

There are six arguments to this subroutine.

GETTOK (IPTR, BUF, LEN, IBUF, ILEN, NCHAR)

IPTR is the current position within the command line. BUF is the command line array, packed two characters per word, of length LEN characters. IBUF is the returned token array, also packed two characters per word, with a maximum length of ILEN words. NCHAR is the number of characters in the returned token. Any unused characters are packed with blanks.

GETTOK starts by packing IBUF with blanks. It then searches BUF from the current position, IPTR, until the beginning of a token is found (the first non-delimiter). Finally, it searches BUF for the next delimiter while updating IPTR and NCHAR, and moving each character into IBUF.

C.5. GETCMD

GETCMD is a simple subroutine which searches a given command table for a match to an input token. There are six arguments.

GETCMD (TBL, LEN, WID, CMD, BUF, NCHAR)

TBL is the input command table of dimensions LEN and WID. CMD returns the index into TBL of the command which matches the token in the array BUF. If no match is found then CMD is set to zero. NCHAR is the number of non-blank characters in the array BUF. Note that WID is one greater than the number of words in each command of TBL. This is because every command has a key which declares the type of parameters, if any, to expect.

The subroutine compares a command in TBL with the token in

BUF on a word by word basis. If it fails then it tries the next command, until finally either finding a match or exhausting the table.

C.6. CNVPAR

CNVPAR is a simple subroutine which converts numerical tokens from ASCII to numerical values. There are six arguments.

CNVPAR (BUF, NCHAR, KEY, IVAL, RVAL, IER)

BUF is the numerical token in ASCII format of length NCHAR characters. The variable KEY which is obtained from the command table, declares the type of numerical value expected where:

KEY = 0	No parameter
KEY = 1	Integer
KEY = 2	Real

IVAL and RVAL are the variables which return the integer or real value respectively. IER is the return code where:

IER = 0	Normal return
IER = 1	Invalid key
IER = 2	Invalid token

The subroutine uses FORTRAN DECODE statements to convert directly from ASCII to numerical form as required.

C.7. KEYBIN

KEYBIN is a subroutine which allows keyboard input during runtime. There are three arguments.

KEYBIN (KNUM, ENTER, IBUF)

KNUM is the current number of characters input from the keyboard. Runtime commands are kept short to save precious time and therefore KNUM is limited to a maximum of 8. ENTER is a logical variable which is set true when the command has been entered. This is determined by finding a line-feed character in the keyboard buffer. The command is returned in the array IBUF.

The subroutine starts by setting ENTER false and calling the Prime subroutine KEYB$$. KEYB$$ returns a one if anything is in the keyboard buffer, and a zero otherwise. If KEYB$$ returns a zero then KEYBIN returns. But, if it returns a one, the Prime subroutine T1IN is called which gets a character from the keyboard buffer. If KEYBIN finds the character "@," then it will erase the array IBUF and the text after the TTY CO label on the screen. A line-feed causes ENTER to be set true. If the character is not an At or line-feed, then it is stored in IBUF, and it is output next to the TTY CO label. The routine then loops back to the KEYB$$ call to check for any remaining characters in the buffer.

C.8. <u>SCREEN</u>

SCREEN is a subroutine which updates the values on the display screen. It also calls the subroutine MAP which updates the display map. There are no subroutine arguments.

The time between display updates, DSPTIM, is variable and can be set by the operator. This is provided because display updating takes a fair amount of time. SCREEN starts by determining whether it is time to update the display or not. If it is, then SCREEN updates the time, heading, pitch, roll, front-axle roll, X and Y locations, speed, steering, laser status, and last rover command. The laser status is the number of bad laser data words, if any, and the last rover command is the last command sent to the rover, usually by the PSA routine. SCREEN also flags any bad value obtained from the rover by putting an asterisk before the respective label. SCREEN finally calls the MAP subroutine and then returns.

C.9. MAP

MAP is a subroutine which updates the display map on the CRT screen. The map displayed is a top view of the rover, or the X-Y plane. The X and Y ranges and origins are variables, and can be set by the operator. These variables are displayed beneath the map for reference during runtime. The screen size of the map is 50 characters in width by 20 characters in height. Because of the poor resolution, the rover is represented by a one digit number which increments modulo-9 every update. If the rover stops or doubles back, the operator will still see the position by the changing number. Note that this map might be too crude to be of any use, but only experience will tell.

MAP starts by quantizing the rover X and Y locations on its limited grid. If the location falls outside the map range, then it is not plotted. The CURSOR subroutine is used to position the cursor, and the digit marker is output. Finally the digit market is updated and MAP returns.

C.10. CURSOR

The CURSOR subroutine is used to position the cursor on the CRT screen. Note that this routine will only work when using an ADM-3 terminal. CURSOR is written in assembler because it is used often in the runtime display updating. There are three arguments:

CURSOR (ROW, COLUMN, IER)

ROW and COLUMN are the desired cursor locations, where the upper left corner is (1,1). IER is a logical return code which is true on an error.

CURSOR first checks the ROW and COLUMN variables for validity, and then adds an offset of 237_8 to each variable. The variables are then packed into a single word with the row in the first byte and the column in the second byte. To position the cursor requires two words. The first is a special code, 115675_8, or an escape followed by an equals. The second is the packed row and column data word. The Prime subroutine TNOUA is called to send this data to the terminal. Finally CURSOR sets IER false and returns.

PART 5

CONCLUSION

A. Summary

The laser triangulation method of hazard detection has been proven feasible by the results obtained during the one laser-one detector system testing. Complete obstacle avoidance was possible with this system at the cost of a slightly conservative path selection ability. This conservatism was very much apparent in a field environment which included slopes. It was because of this inability of the one laser-one detector system to interpret slopes that the elevation scanning system evolved.

Simulation studies indicate that the elevation scanning system does provide enough data for an accurate slope appraisal. Currently, all rover systems are being modified to implement this new hazard detection system.

New realtime software has been written on a Prime minicomputer. The hazard detection and obstacle avoidance software development work has been done using the simulator. However, the realtime support software has only had some limited testing since none of the required hardware is working yet.

This support software has been designed to provide the user with a simple interactive environment. The user can change and display parameters, control the rover either manually or autonomously, and even run as many tests as desired without ever halting the program. Important runtime values are displayed and all raw data is saved for post-run analysis.

B. Future Work

Once the problems with the Prime test interface are solved, it is important that the lower level realtime software be completely tested. After the test interface is understood, the realtime interface can be designed and constructed.

The first realtime laser data will come from the dynamic test platform. Initial testing should be done with a static platform. Only when it appears that the hazard detection software can accurately interpret any scene can the rover be tested under realtime control.

Programs to analyze the post-run data should be developed immediately since they will be necessary when testing the hazard detection software. Running off-line, the post-processor routines can use computer graphics to help display data.

Although first testing should use a dedicated processor, it may be possible to allow other users on the system during realtime testing. This would require that our operating system be kept up to date with the normal operating system and that it have a low probability of crashing the system. Also, although unlikely because of time constraints, the possibility of realtime graphics should be investigated.

PART 6

LITERATURE CITED

1. Krajewski, Marjan, The Development and Evaluation of a Short Range Path Selection System for an Autonomous Planetary Rover, RPI Master's Report, RPI, Troy, N.Y., 1976.

2. Sadeghi, Tahun, Development of a Path Selection Program for the Mars Rover, RPI Progress Report, RPI, Troy, N.Y., 1977.

3. Dunn, Paul, Progress in Path Selection Algorithms, RPI Progress Report, RPI, Troy, N.Y., 1977.

4. Meshach, Bill, Elevation Scanning/Multi-Detector Hazard Detection System: Pulsed Laser and Photo Detector System, RPI Master's Project, RPI, Troy, N.Y., 1978.

5. Craig, John, Design and Implementation of the Laser Scanning/ Multi-Detector Controller for Hazard Detection System, RPI Master's Project, RPI, Troy, N.Y., 1978.

6. Prime Software Documentation, Prime Computer, Inc., Framingham, Mass., 1978.

APPENDIX A

Azimuth Data Word Formats

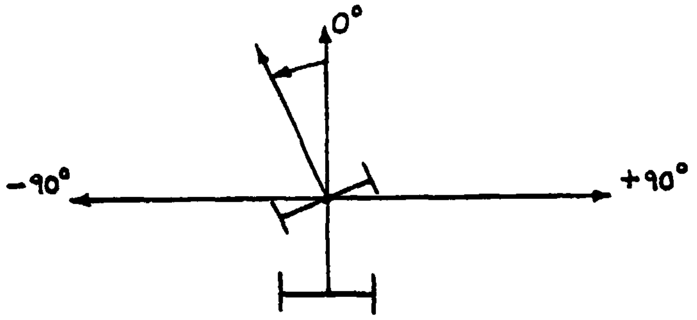
1. **Steering Angle:**

Range: -90° to +90°

Data Format:

```
15      12 11                0
| 0 0 0 0 |     A/D Data     |
```

A/D Data	Steering Angle
-2047	-90°
0	0°
+2047	+90°



2. **Tachometers:**

Data Format:

```
15      12 11                0
| 0 0 0 0 |     A/D Data     |
```

A/D Data	Wheel Speed
-2047	Full Reverse
0	Stopped
+2047	Full Forward

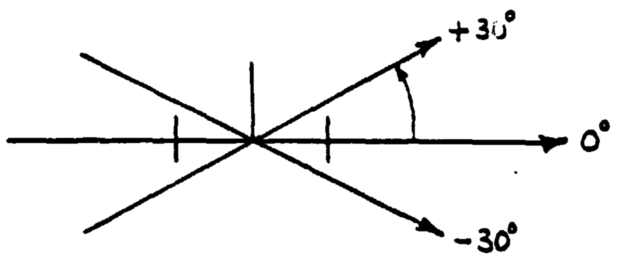
3. **Front Axle Roll Angle:**

Range: -30° to +30°

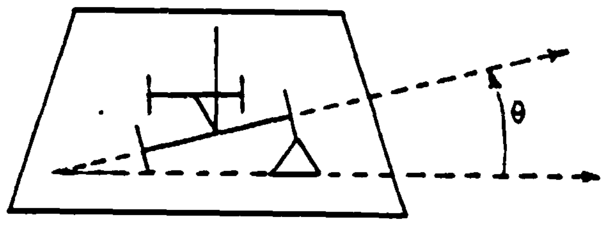
Data Format:

15 – 12	11 – 0
0 0 0 0	A/D Data

A/D Data	Front Axle Roll Angle
-2047	-30°
0	0°
+2047	+30°



Front View



Front Axle Roll

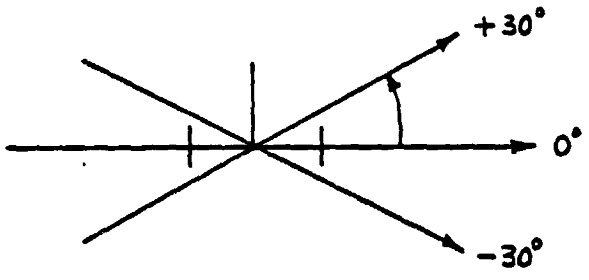
4. <u>Roll Angle</u>:

Range: -30° to +30°

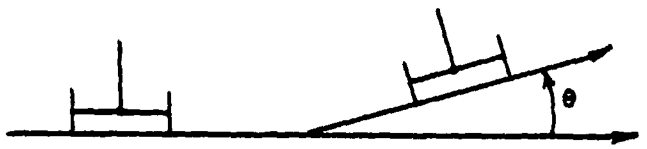
Data Format:

15	12	11	0
0 0 0 0		A/D Data	

<u>A/D Data</u>	<u>Roll Angle</u>
-2047	-30°
0	0°
+2047	+30°



Front View



Roll

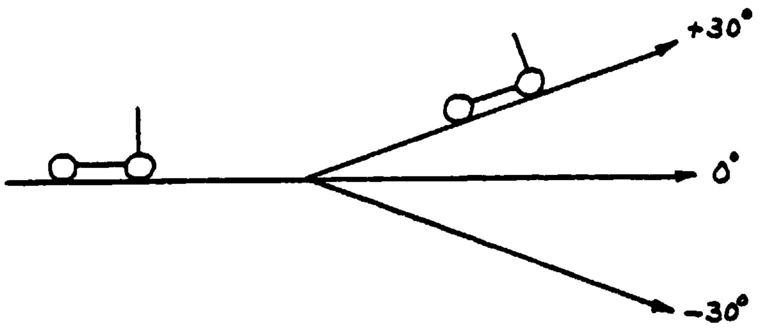
5. **Pitch Angle:**

Range: -30° to +30°

Data Format:

15 12	11 0
0 0 0 0	A/D Data

A/D Data	Pitch Angle
-2047	-30°
0	0°
+2047	+30°

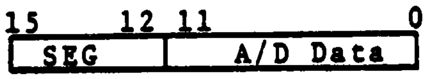


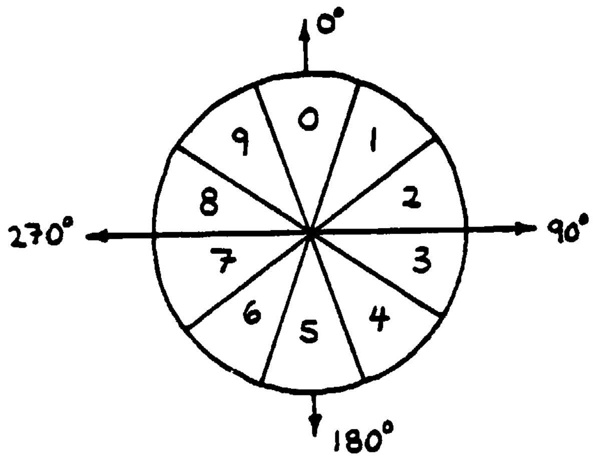
Pitch

6. **Heading Angle:**

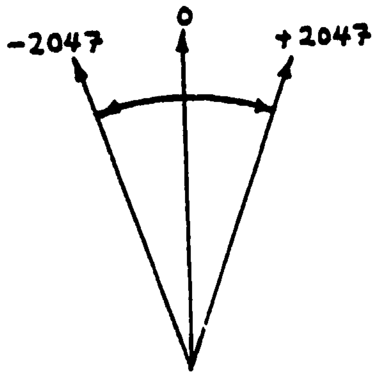
Range: 0° to 360°

Data Format:

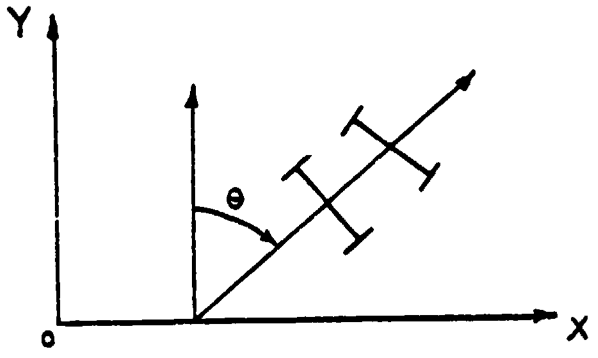




SEG Number

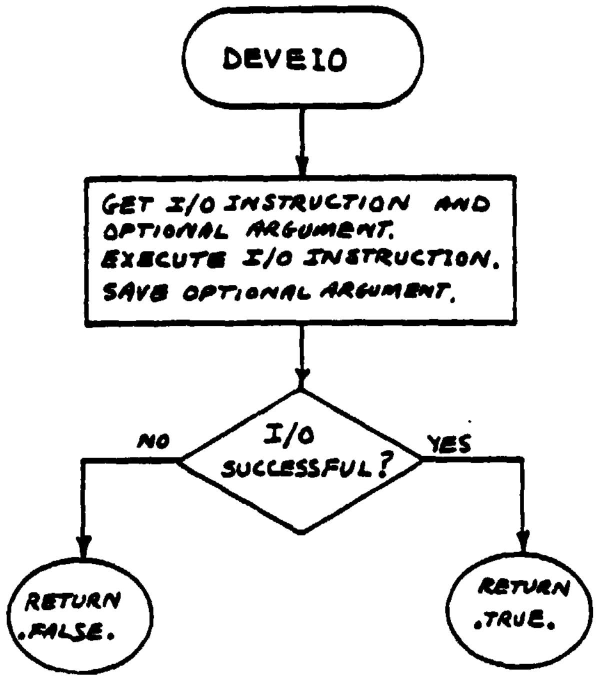


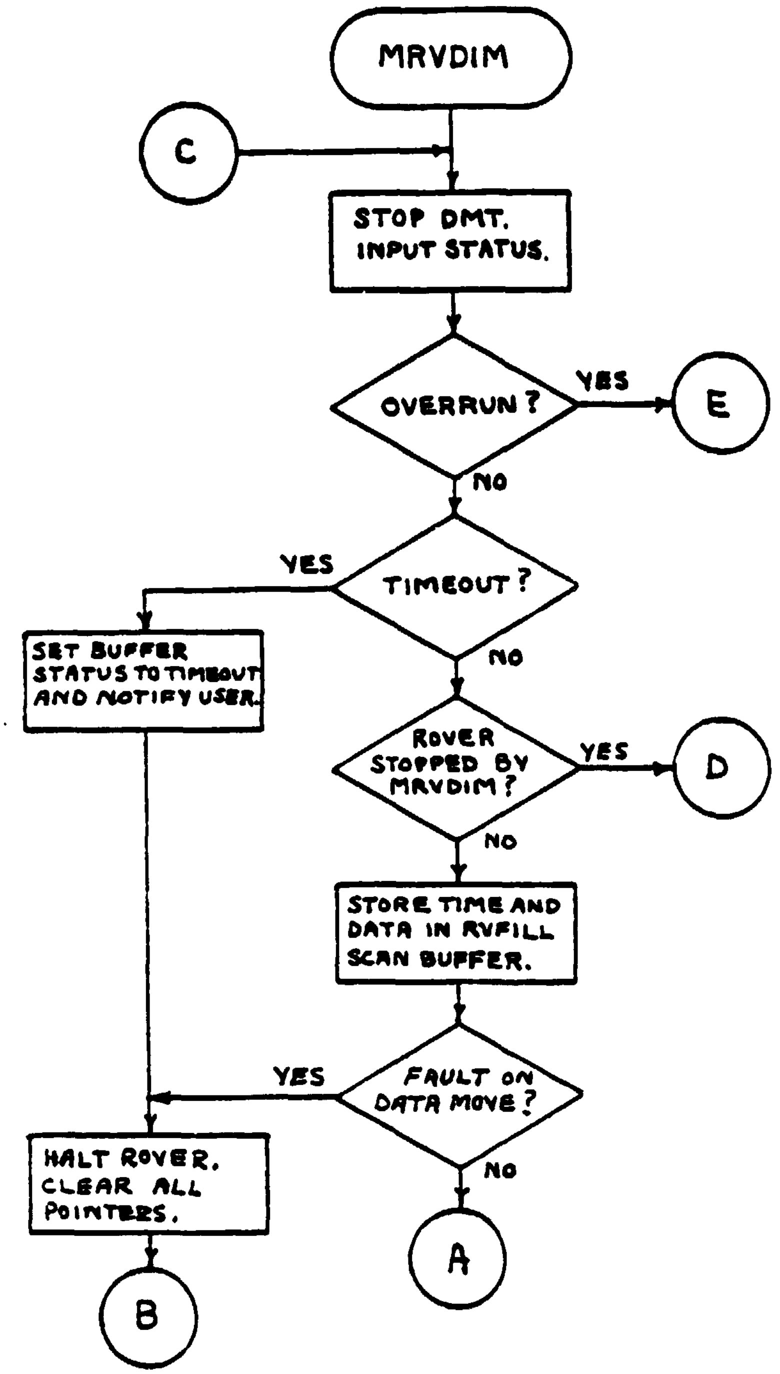
A/D Data per Segment

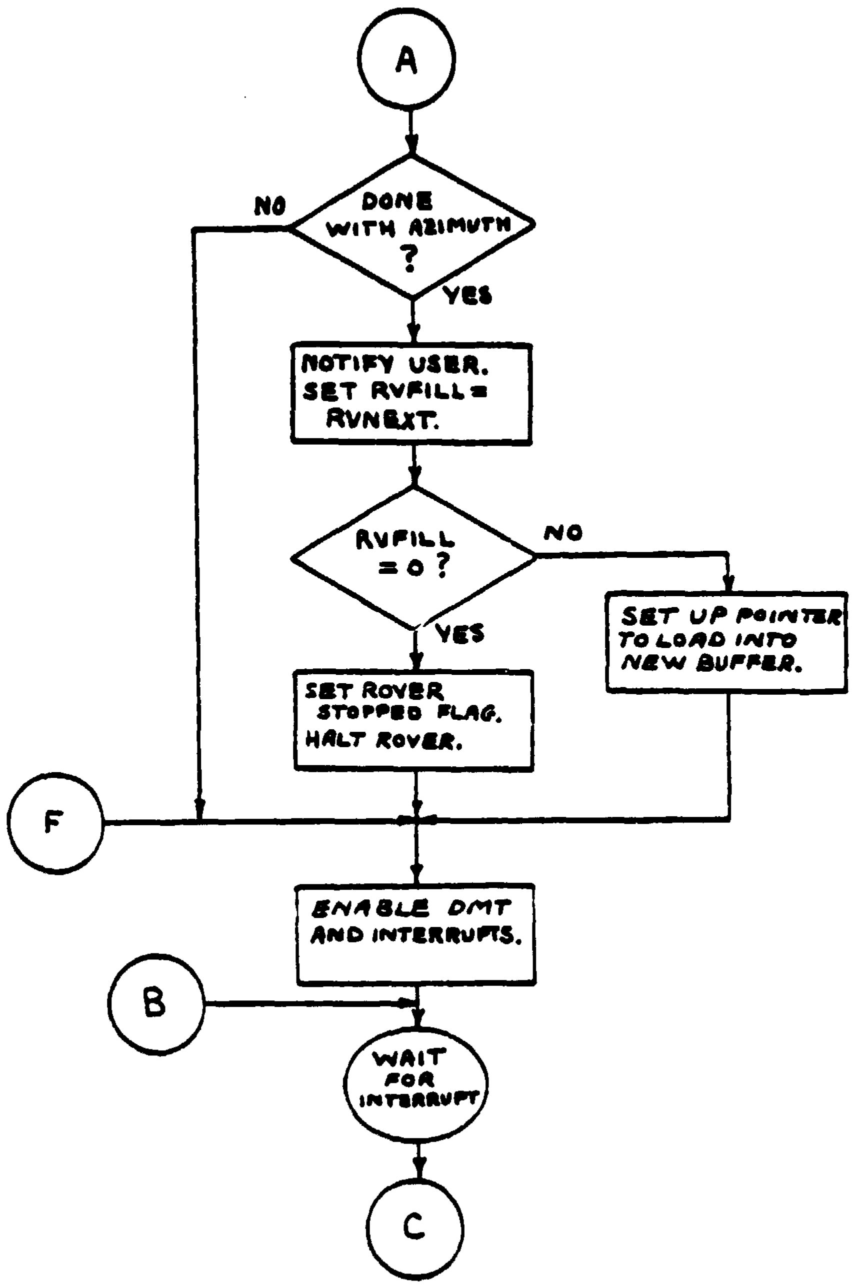


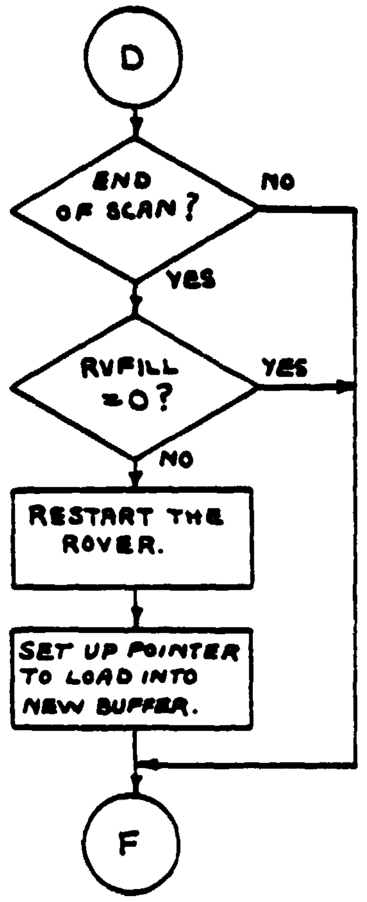
Heading

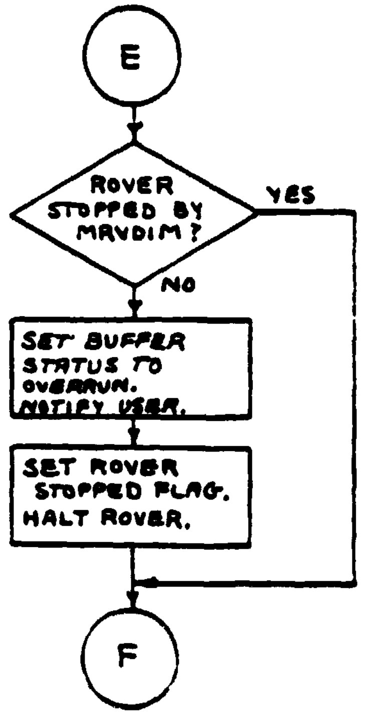
APPENDIX B

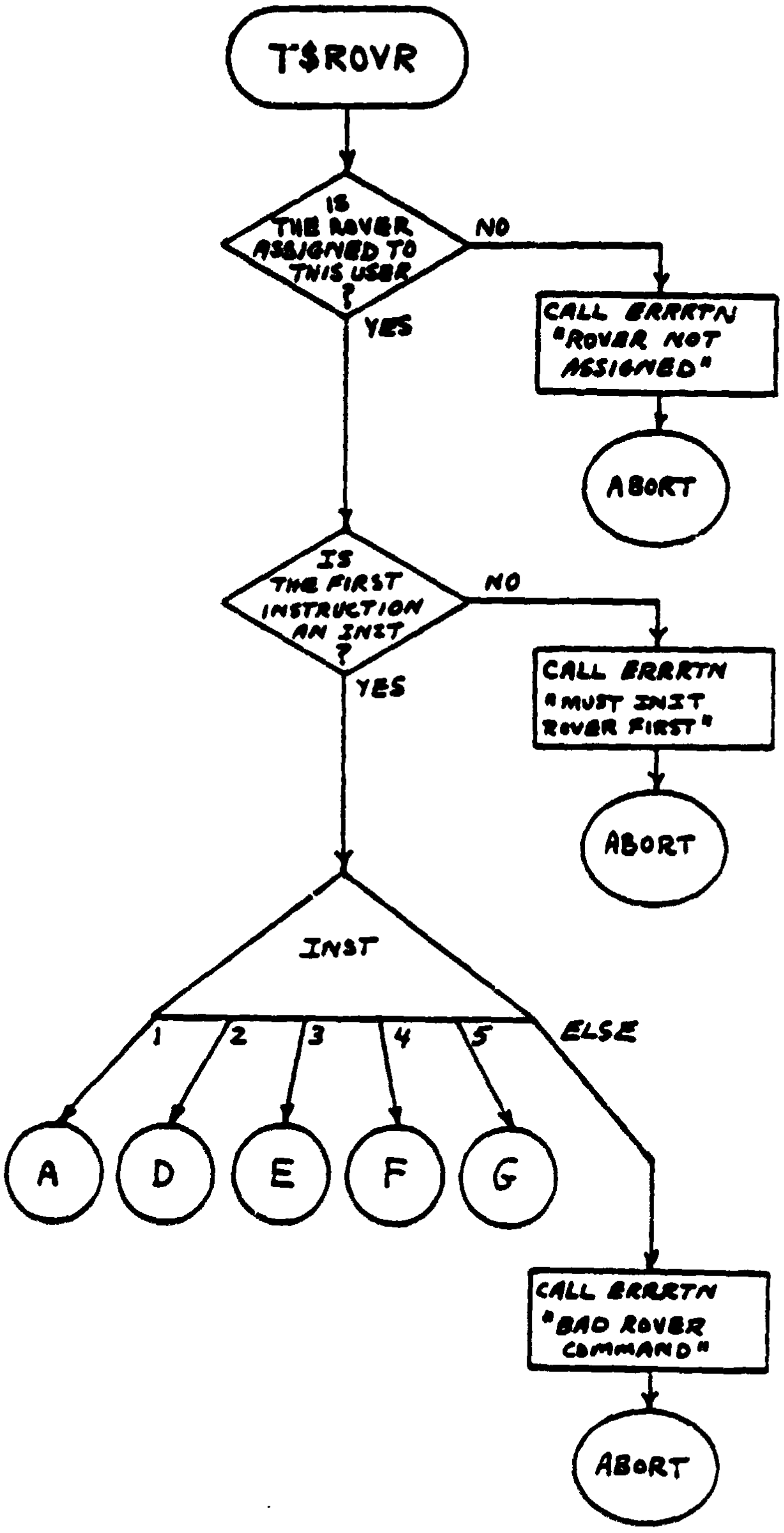

DEVEIO
GET I/O INSTRUCTION AND OPTIONAL ARGUMENT.
EXECUTE I/O INSTRUCTION.
SAVE OPTIONAL ARGUMENT.
I/O SUCCESSFUL?
NO
YES
RETURN .FALSE.
RETURN .TRUE.

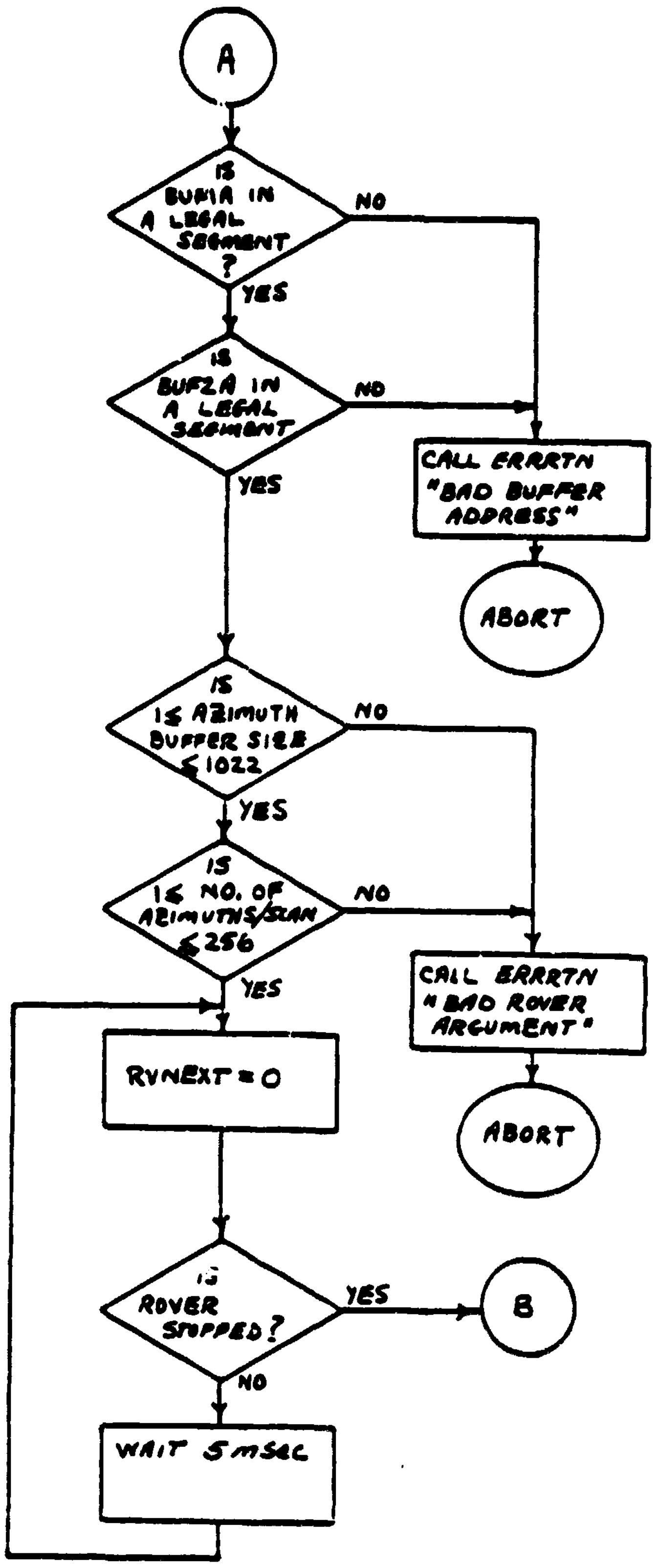

MRVDIM
C
STOP DMT.
INPUT STATUS.
OVERRUN ?
YES
E
NO
TIMEOUT ?
YES
SET BUFFER STATUS TO TIMEOUT AND NOTIFY USER.
NO
ROVER STOPPED BY MRVDIM ?
YES
D
NO
STORE TIME AND DATA IN RVFILL SCAN BUFFER.
FAULT ON DATA MOVE ?
YES
NO
HALT ROVER. CLEAR ALL POINTERS.
B
A

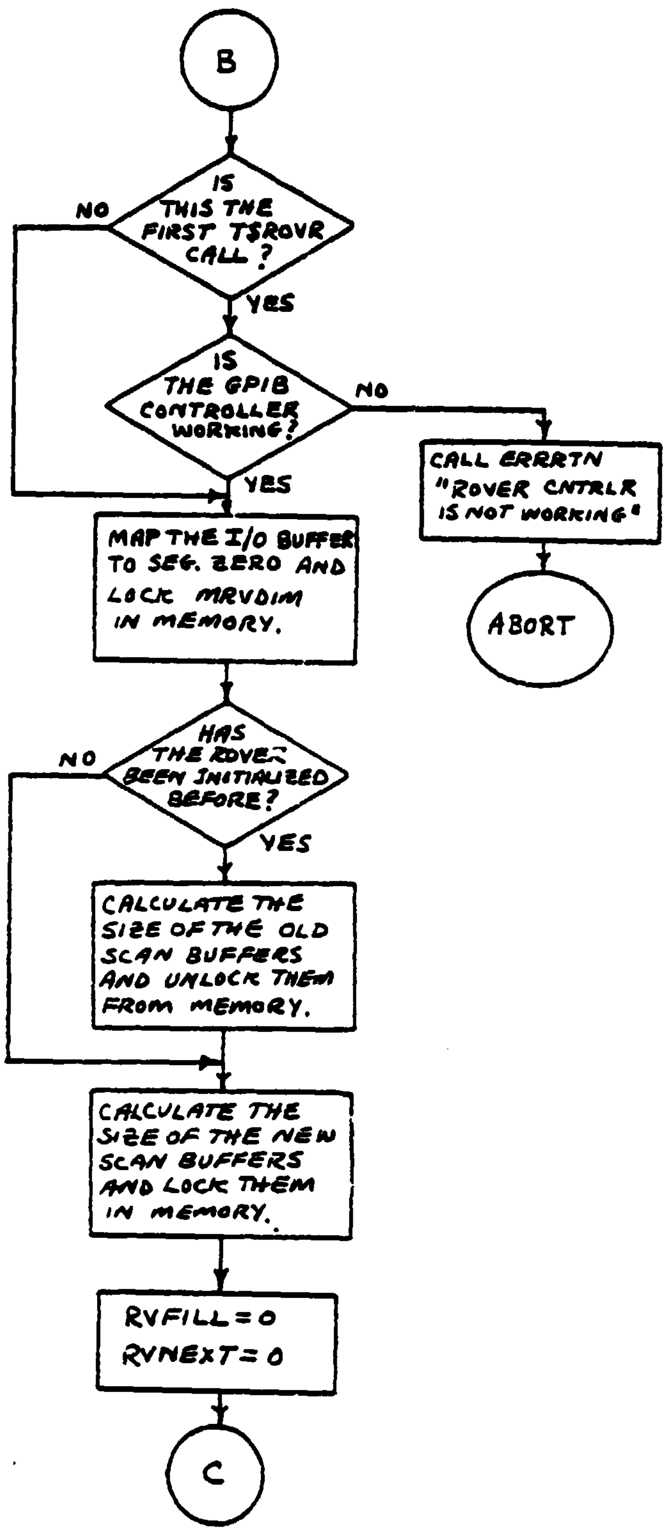

A
DONE WITH AZIMUTH ?
NO
YES
NOTIFY USER. SET RVFILL = RVNEXT.
RVFILL = 0 ?
NO
YES
SET UP POINTER TO LOAD INTO NEW BUFFER.
SET ROVER STOPPED FLAG. HALT ROVER.
F
ENABLE DMT AND INTERRUPTS.
B
WAIT FOR INTERRUPT
C

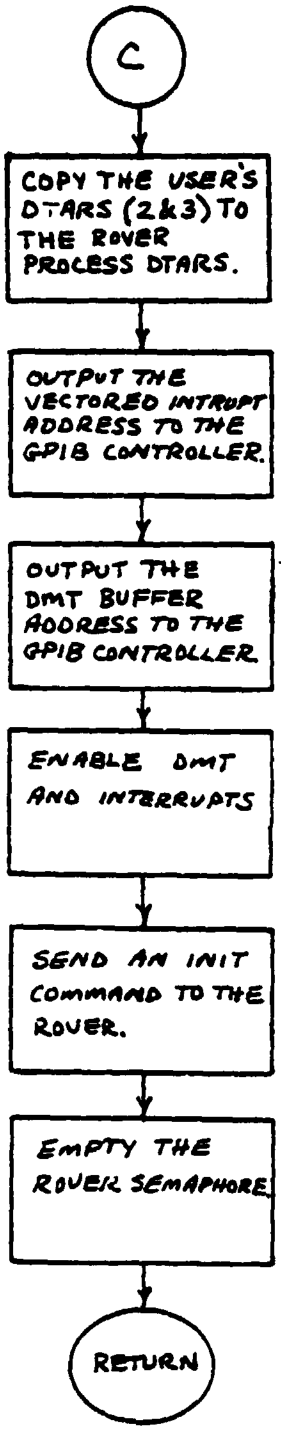

D
END OF SCAN ?
NO
YES
RVFILL =0?
YES
NO
RESTART THE ROVER.
SET UP POINTER TO LOAD INTO NEW BUFFER.
F

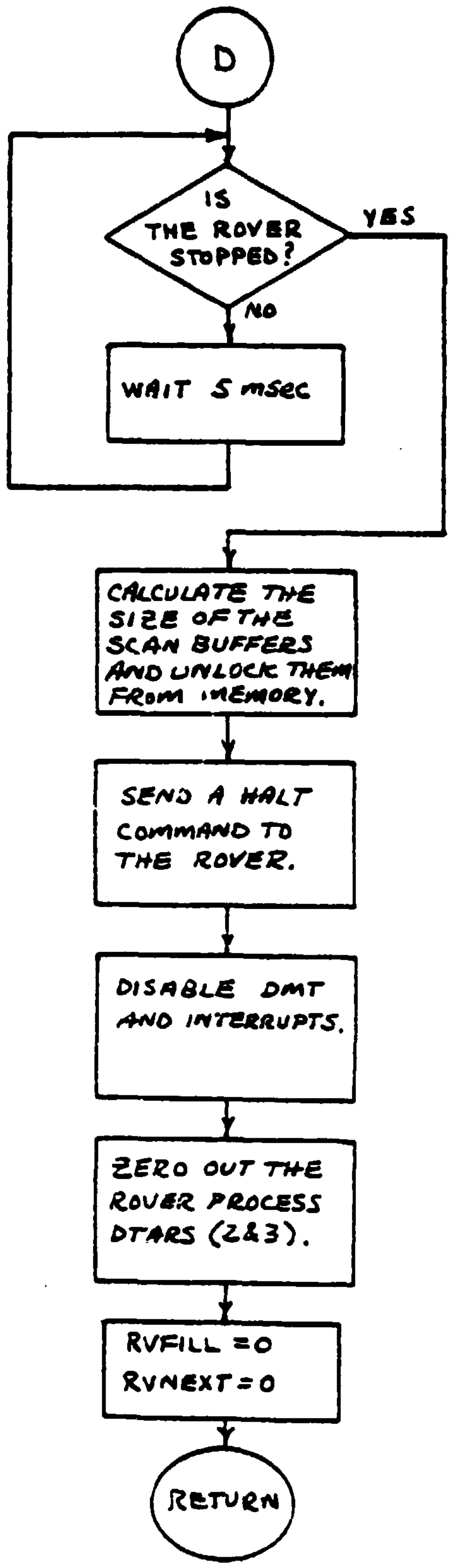

E
ROVER STOPPED BY MRVDIM ?
YES
NO
SET BUFFER STATUS TO OVERRUN. NOTIFY USER.
SET ROVER STOPPED FLAG. HALT ROVER.
F

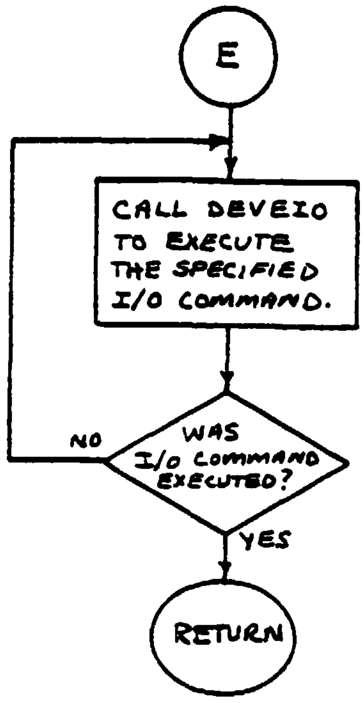

T$ROVR
IS THE ROVER ASSIGNED TO THIS USER ?
NO
YES
CALL ERRRTN "ROVER NOT ASSIGNED"
ABORT
IS THE FIRST INSTRUCTION AN INIT ?
NO
YES
CALL ERRRTN "MUST INIT ROVER FIRST"
ABORT
INST
1
2
3
4
5
ELSE
A
D
E
F
G
CALL ERRRTN "BAD ROVER COMMAND"
ABORT

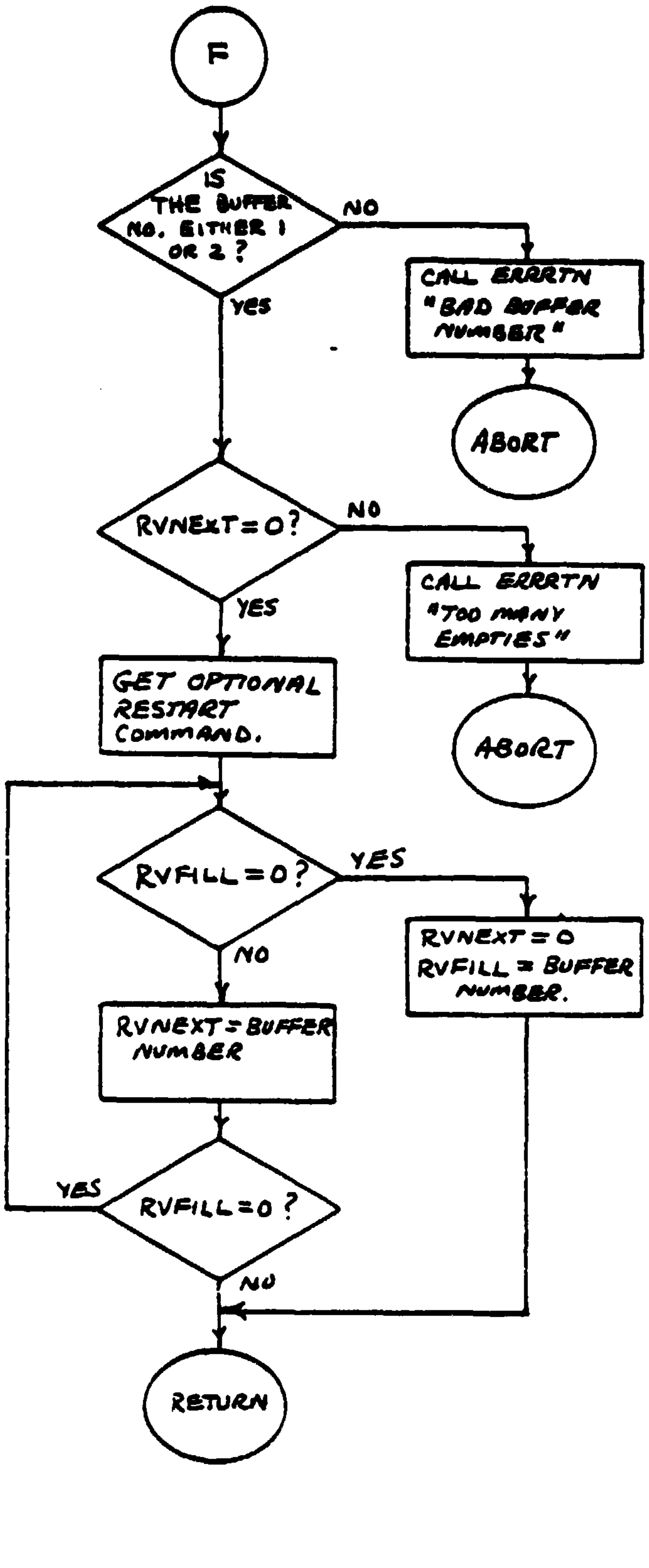

A
IS BUF1A IN A LEGAL SEGMENT ?
NO
YES
IS BUF2A IN A LEGAL SEGMENT
NO
YES
CALL ERRRTN "BAD BUFFER ADDRESS"
ABORT
IS 1 ≤ AZIMUTH BUFFER SIZE ≤ 1022
NO
YES
IS 1 ≤ NO. OF AZIMUTHS/SCAN ≤ 256
NO
YES
CALL ERRRTN "BAD ROVER ARGUMENT"
ABORT
RVNEXT = 0
IS ROVER STOPPED?
YES
B
NO
WAIT 5 mSEC

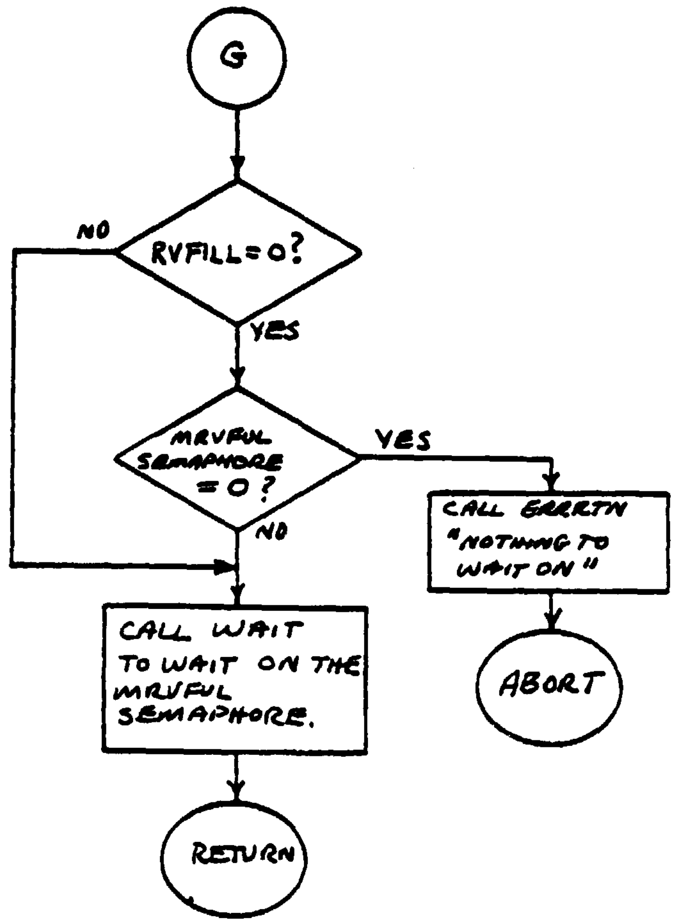

B
IS THIS THE FIRST T$ROVR CALL?
NO
YES
IS THE GPIB CONTROLLER WORKING?
NO
YES
CALL ERRRTN "ROVER CNTRLR IS NOT WORKING"
ABORT
MAP THE I/O BUFFER TO SEG. ZERO AND LOCK MRVDIM IN MEMORY.
HAS THE ROVER BEEN INITIALIZED BEFORE?
NO
YES
CALCULATE THE SIZE OF THE OLD SCAN BUFFERS AND UNLOCK THEM FROM MEMORY.
CALCULATE THE SIZE OF THE NEW SCAN BUFFERS AND LOCK THEM IN MEMORY.
RVFILL=0
RVNEXT=0
C

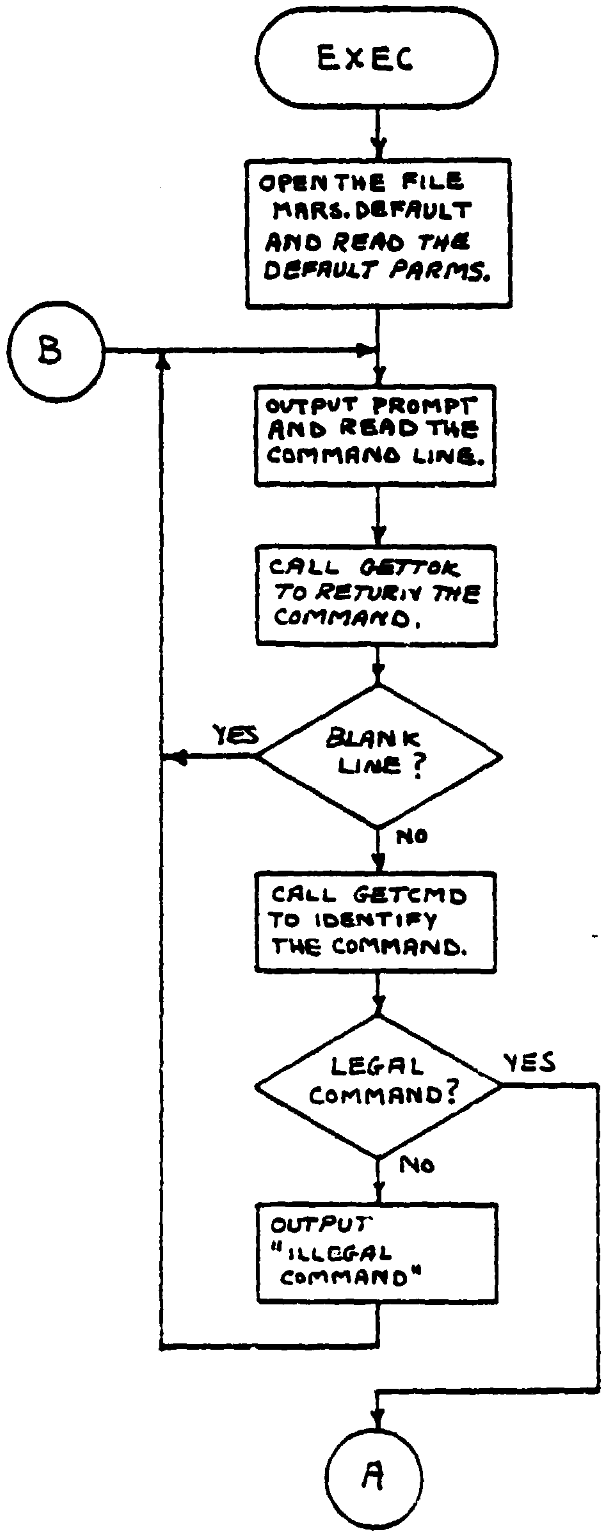

C
COPY THE USER'S DTARS (2&3) TO THE ROVER PROCESS DTARS.
OUTPUT THE VECTORED INTRUPT ADDRESS TO THE GPIB CONTROLLER.
OUTPUT THE DMT BUFFER ADDRESS TO THE GPIB CONTROLLER.
ENABLE DMT AND INTERRUPTS
SEND AN INIT COMMAND TO THE ROVER.
EMPTY THE ROVER SEMAPHORE.
RETURN

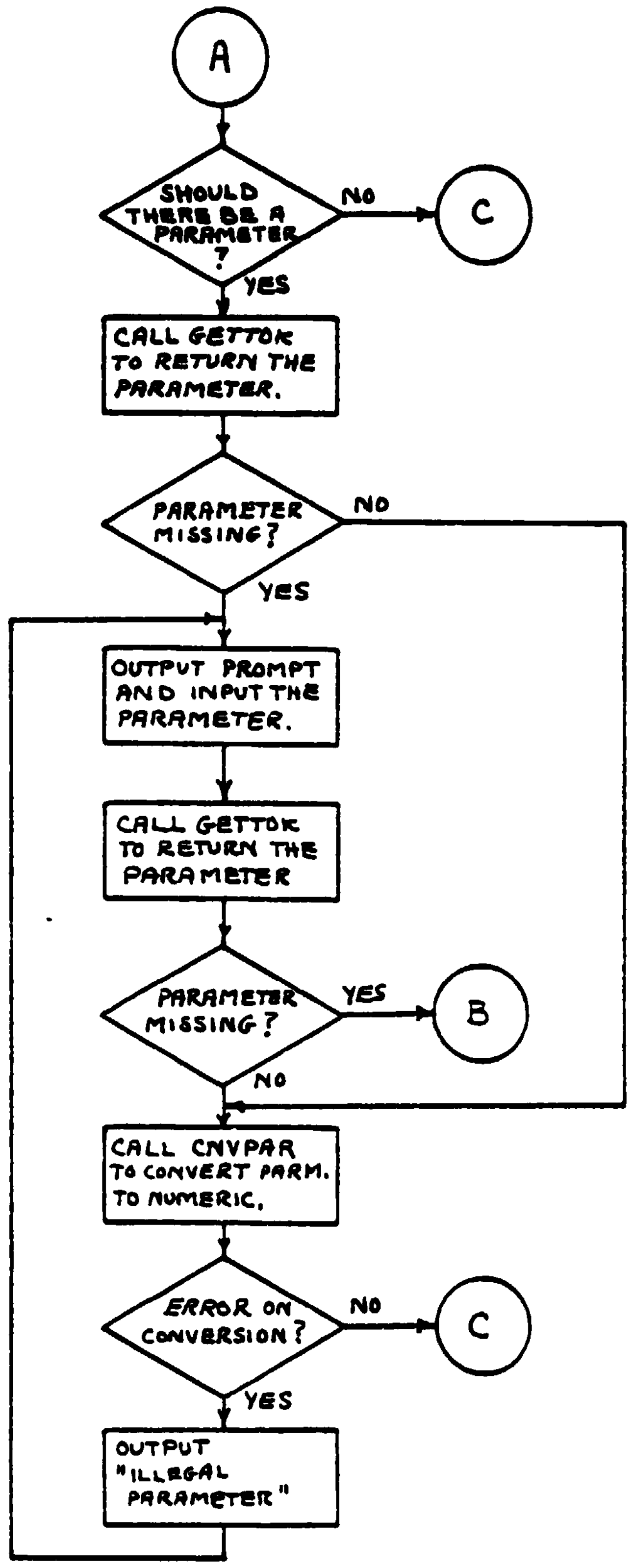

D
IS THE ROVER STOPPED?
YES
NO
WAIT 5 msec
CALCULATE THE SIZE OF THE SCAN BUFFERS AND UNLOCK THEM FROM MEMORY.
SEND A HALT COMMAND TO THE ROVER.
DISABLE DMT AND INTERRUPTS.
ZERO OUT THE ROVER PROCESS DTARS (2&3).
RVFILL =0
RVNEXT=0
RETURN

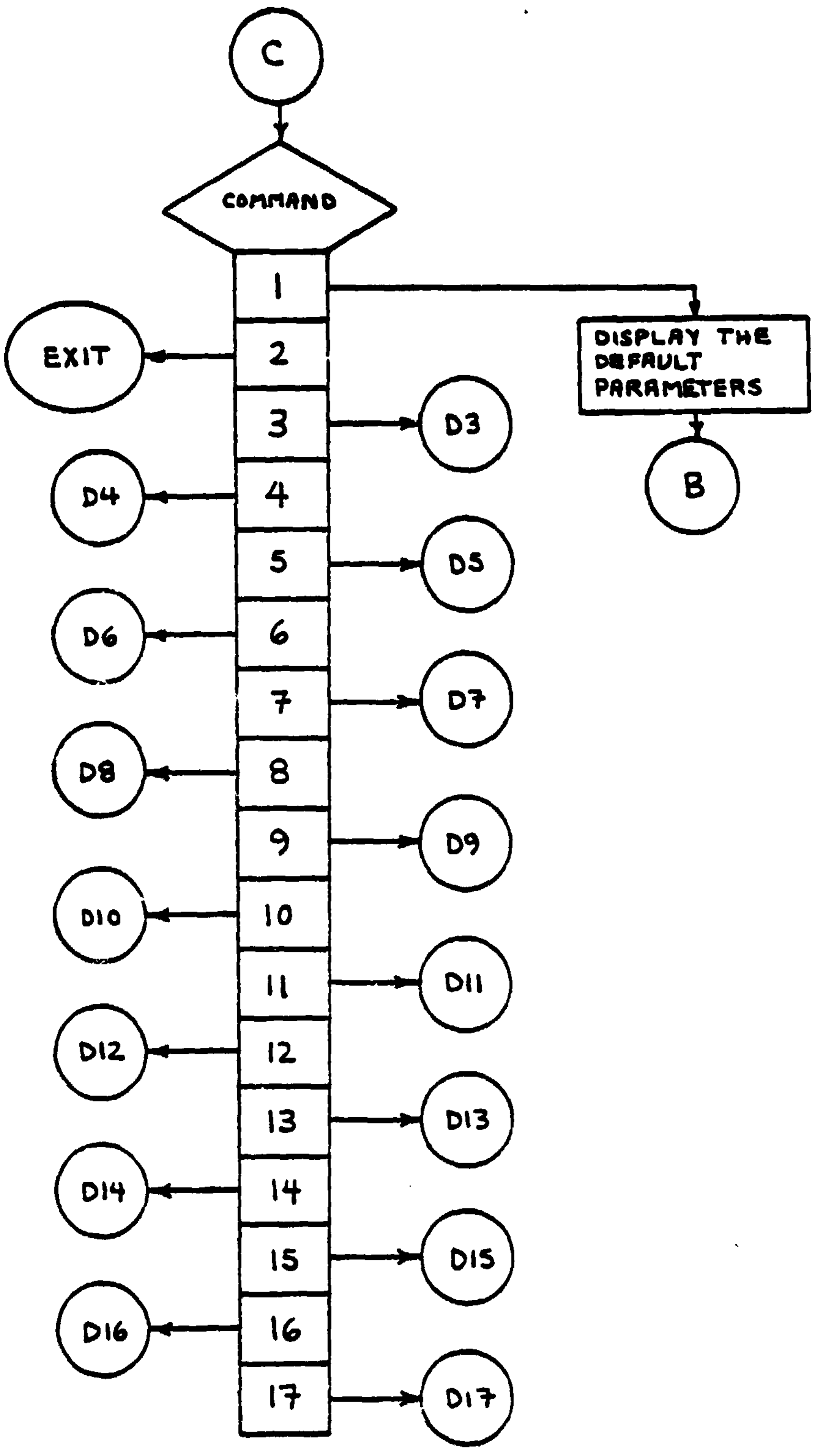

E
CALL DEVEIO TO EXECUTE THE SPECIFIED I/O COMMAND.
WAS I/O COMMAND EXECUTED?
NO
YES
RETURN

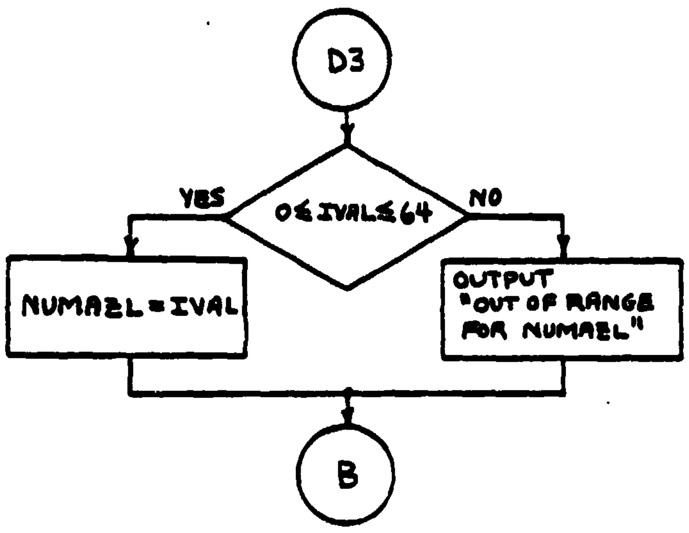

F
IS THE BUFFER NO. EITHER 1 OR 2 ?
NO
YES
CALL ERRRTN "BAD BUFFER NUMBER"
ABORT
RVNEXT=0?
NO
YES
CALL ERRRTN "TOO MANY EMPTIES"
ABORT
GET OPTIONAL RESTART COMMAND.
RVFILL=0?
YES
NO
RVNEXT=0 RVFILL=BUFFER NUMBER.
RVNEXT=BUFFER NUMBER
YES
RVFILL=0 ?
NO
RETURN

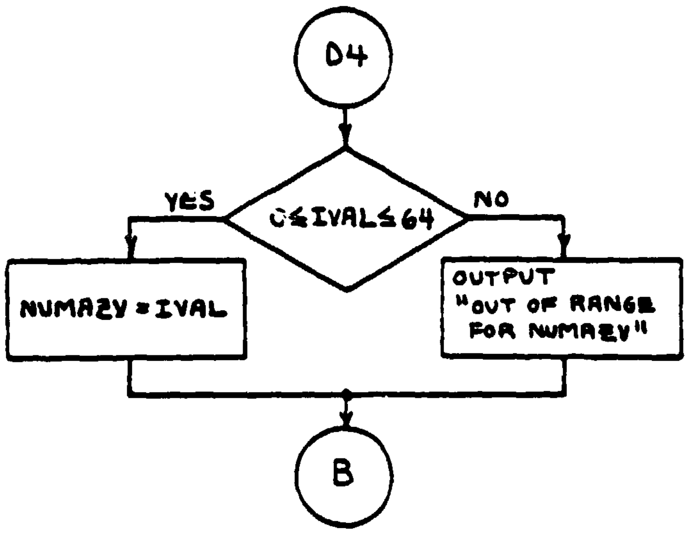

G
RVFILL=0?
NO
YES
MRVFUL SEMAPHORE =0?
YES
NO
CALL ERRRTN "NOTHING TO WAIT ON"
ABORT
CALL WAIT TO WAIT ON THE MRVFUL SEMAPHORE.
RETURN

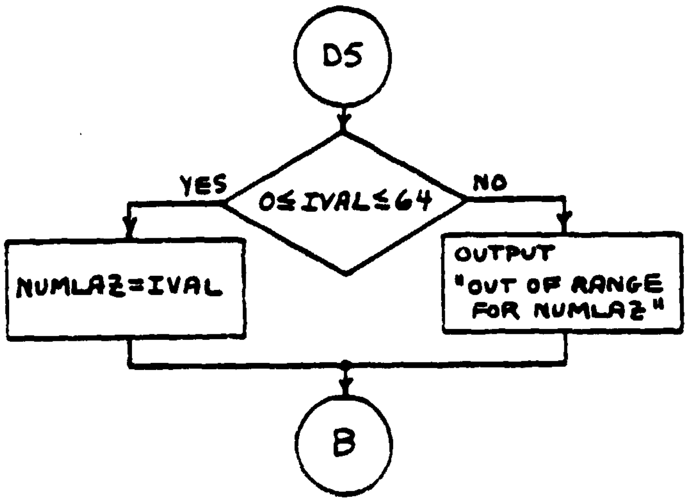

EXEC
OPEN THE FILE MARS.DEFAULT AND READ THE DEFAULT PARMS.
B
OUTPUT PROMPT AND READ THE COMMAND LINE.
CALL GETTOK TO RETURN THE COMMAND.
BLANK LINE ?
YES
NO
CALL GETCMD TO IDENTIFY THE COMMAND.
LEGAL COMMAND?
YES
NO
OUTPUT "ILLEGAL COMMAND"
A

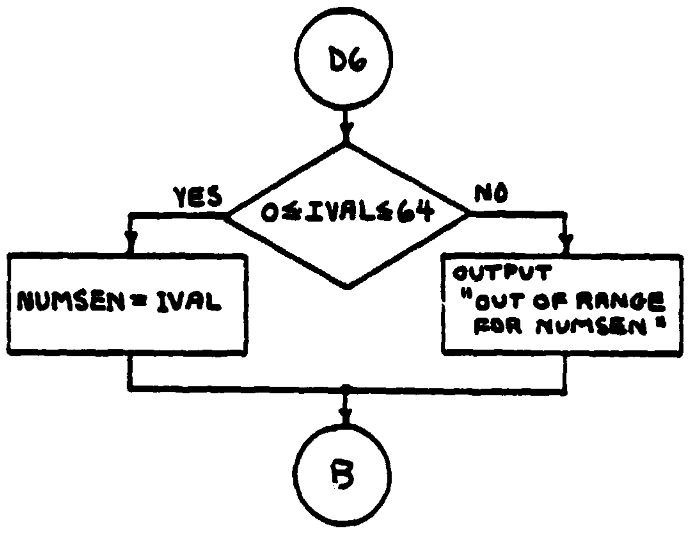

A
SHOULD THERE BE A PARAMETER ?
NO
C
YES
CALL GETTOK TO RETURN THE PARAMETER.
PARAMETER MISSING?
NO
YES
OUTPUT PROMPT AND INPUT THE PARAMETER.
CALL GETTOK TO RETURN THE PARAMETER
PARAMETER MISSING?
YES
B
NO
CALL CNVPAR TO CONVERT PARM. TO NUMERIC.
ERROR ON CONVERSION?
NO
C
YES
OUTPUT "ILLEGAL PARAMETER"

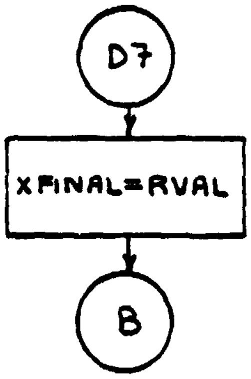

C
COMMAND
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
EXIT
DISPLAY THE DEFAULT PARAMETERS
B
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
D14
D15
D16
D17

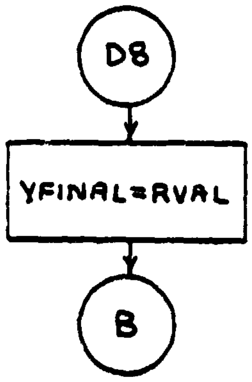

D3
0≤IVAL≤64
YES
NO
NUMAZL=IVAL
OUTPUT "OUT OF RANGE FOR NUMAEL"
B

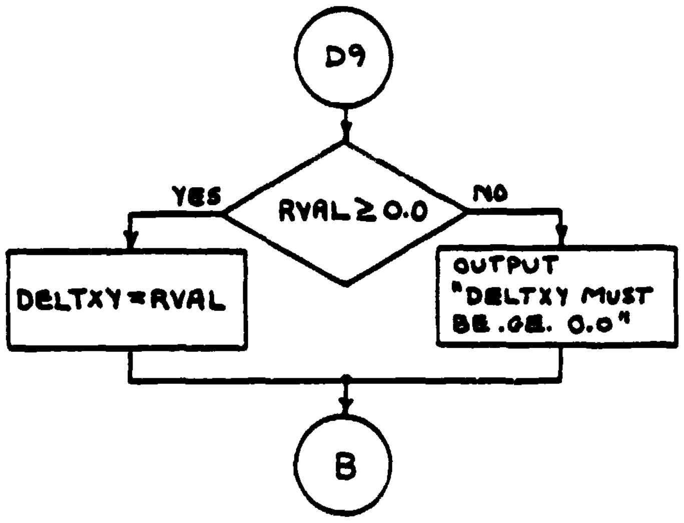

D4
0≤IVAL≤64
YES
NO
NUMAZV=IVAL
OUTPUT "OUT OF RANGE FOR NUMAZV"
B

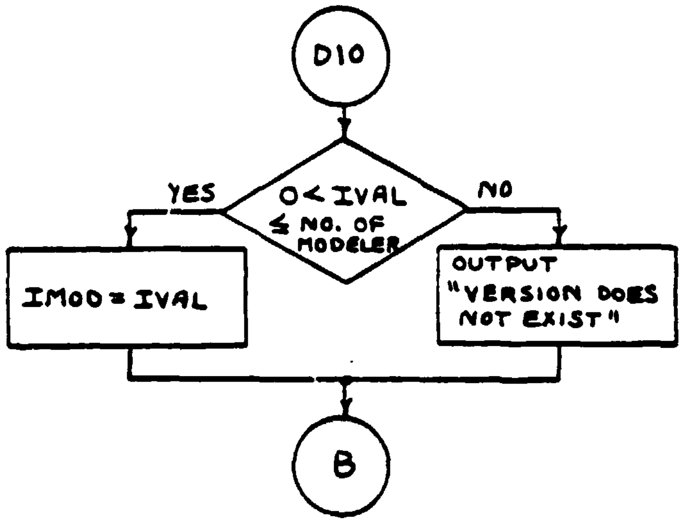

D5
0≤IVAL≤64
YES
NO
NUMLAZ=IVAL
OUTPUT "OUT OF RANGE FOR NUMLAZ"
B

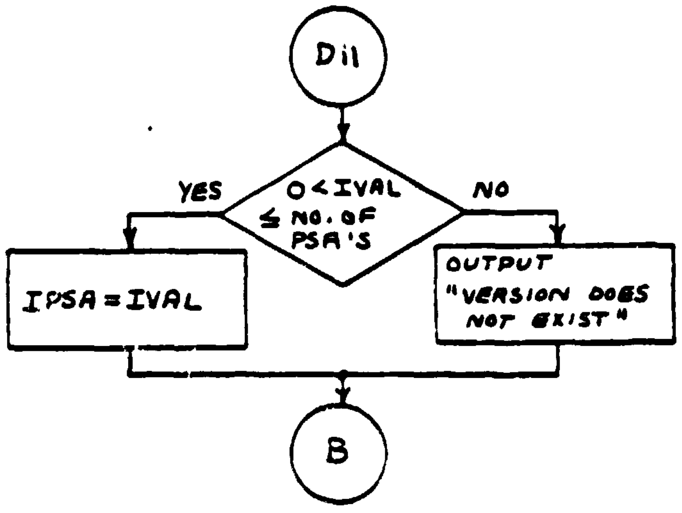

D6
0≤IVAL≤64
YES
NO
NUMSEN = IVAL
OUTPUT "OUT OF RANGE FOR NUMSEN"
B

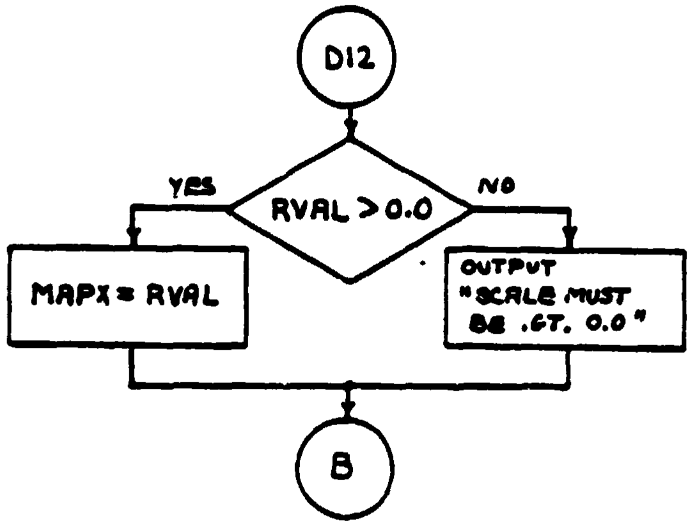

D7
XFINAL=RVAL
B

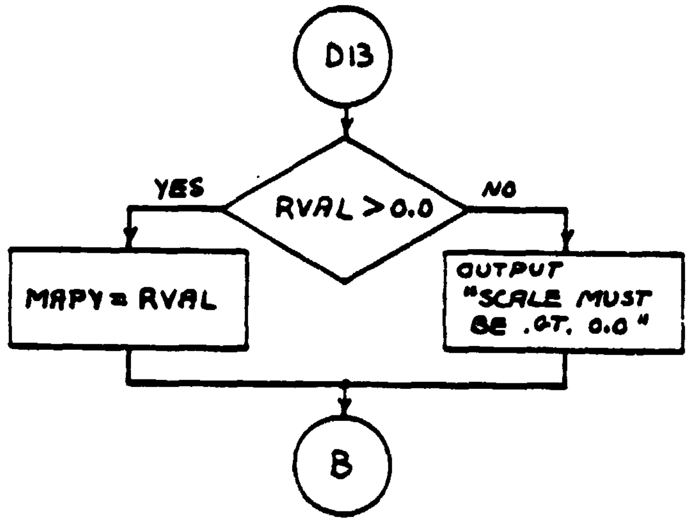

D8
YFINAL=RVAL
B

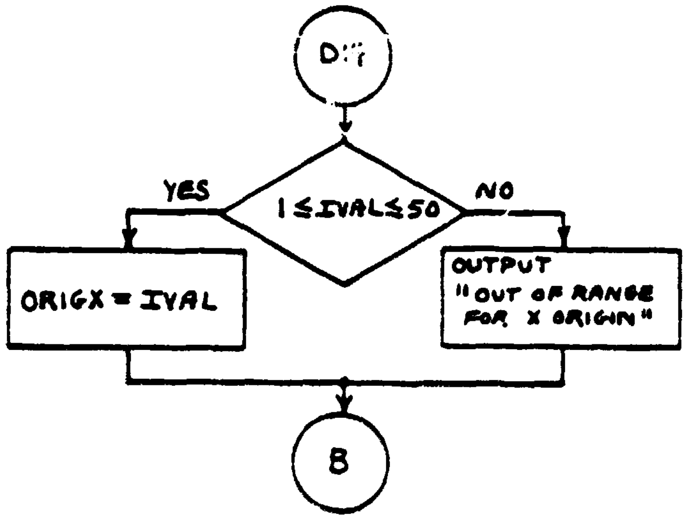

D9
RVAL ≥ 0.0
YES
NO
DELTXY = RVAL
OUTPUT "DELTXY MUST BE .GE. 0.0"
B

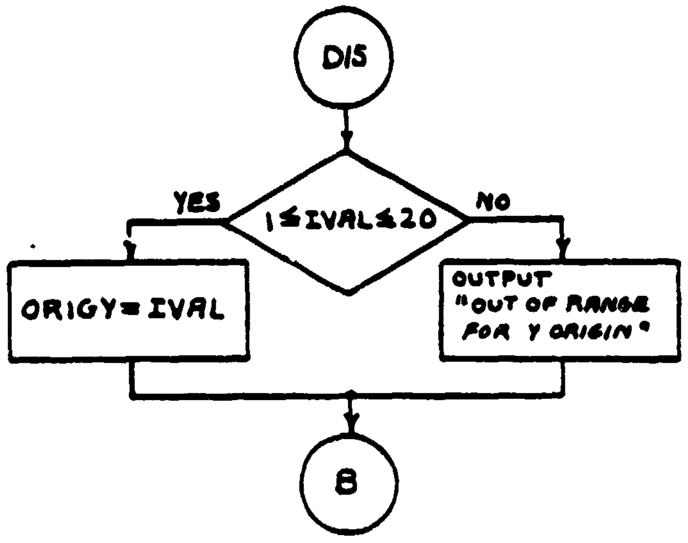

D10
0 < IVAL ≤ NO. OF MODELER
YES
NO
IMOD = IVAL
OUTPUT "VERSION DOES NOT EXIST"
B

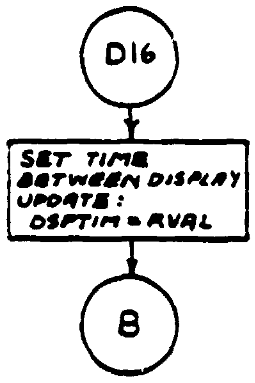

D11
0 < IVAL ≤ NO. OF PSA'S
YES
NO
IPSA = IVAL
OUTPUT "VERSION DOES NOT EXIST"
B

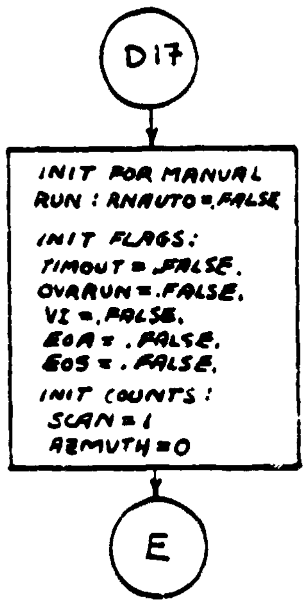

D12
RVAL > 0.0
YES
NO
MAPX = RVAL
OUTPUT "SCALE MUST BE .GT. 0.0"
B

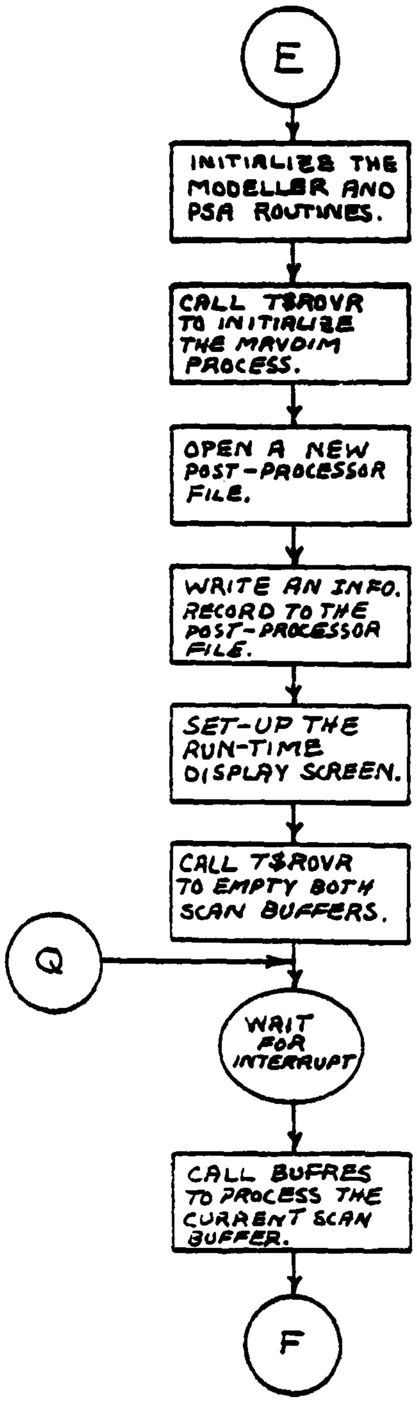

D13
RVAL > 0.0
YES
NO
MAPY = RVAL
OUTPUT "SCALE MUST BE .GT. 0.0"
B


D14
1 ≤ IVAL ≤ 50
YES
NO
ORIGX = IVAL
OUTPUT "OUT OF RANGE FOR X ORIGIN"
B

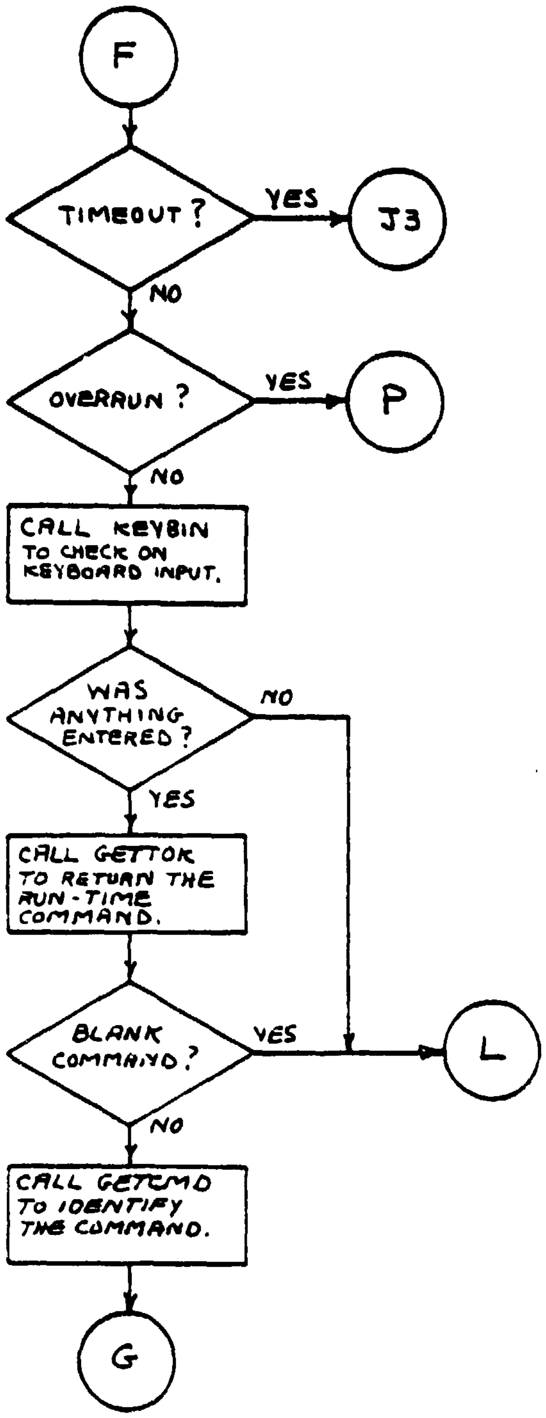

D15
1≤IVAL≤20
YES
NO
ORIGY=IVAL
OUTPUT "OUT OF RANGE FOR Y ORIGIN"
B

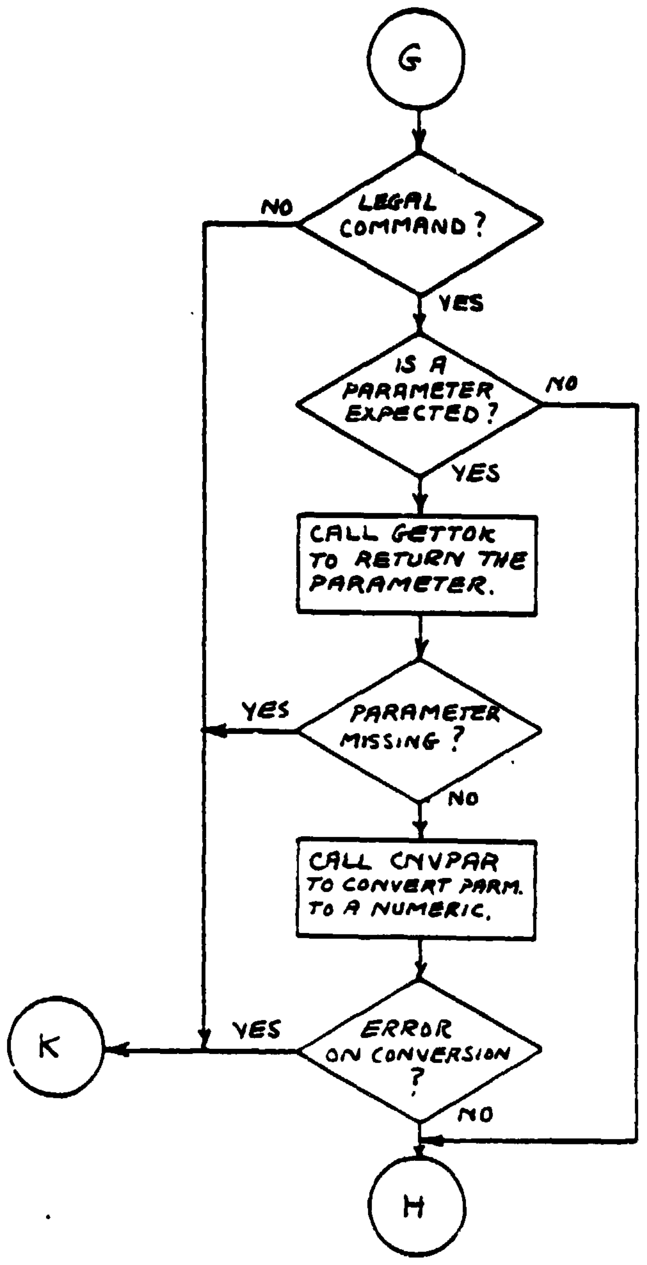

D16
SET TIME BETWEEN DISPLAY UPDATE :
DSPTIM = RVAL
B

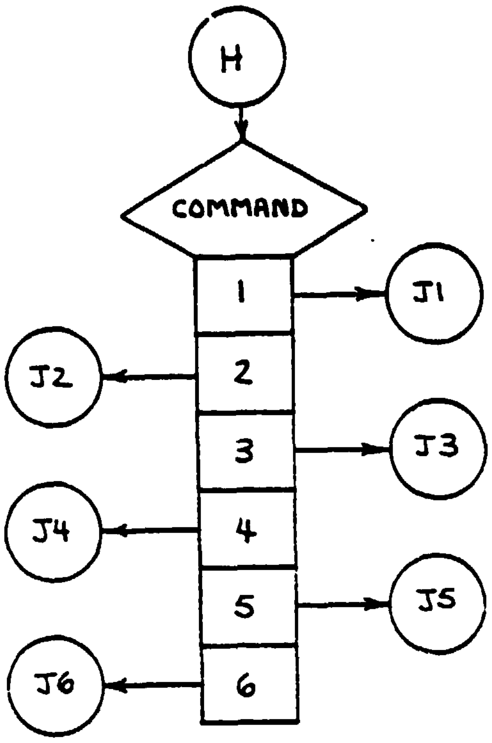

D17
INIT FOR MANUAL RUN : RNAUTO=.FALSE.
INIT FLAGS:
TIMOUT = .FALSE.
OVRRUN = .FALSE.
VI = .FALSE.
EOA = .FALSE.
EOS = .FALSE.
INIT COUNTS :
SCAN = 1
AZMUTH = 0
E

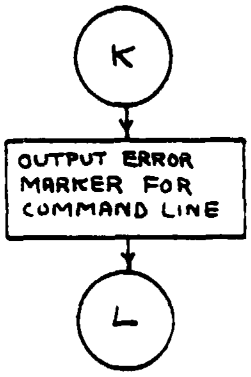

E
INITIALIZE THE MODELLER AND PSA ROUTINES.
CALL T$ROVR TO INITIALIZE THE MAVDIM PROCESS.
OPEN A NEW POST-PROCESSOR FILE.
WRITE AN INFO. RECORD TO THE POST-PROCESSOR FILE.
SET-UP THE RUN-TIME DISPLAY SCREEN.
CALL T$ROVR TO EMPTY BOTH SCAN BUFFERS.
Q
WAIT FOR INTERRUPT
CALL BUFRES TO PROCESS THE CURRENT SCAN BUFFER.
F

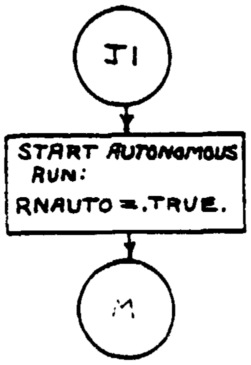
C-2

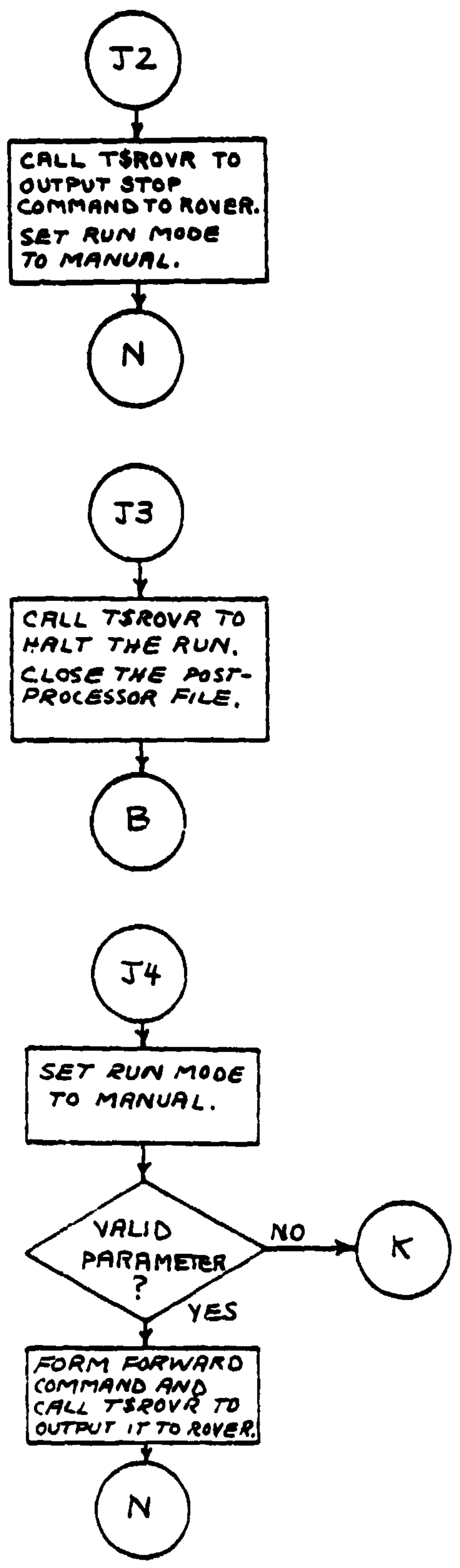

F
TIMEOUT ?
YES
J3
NO
OVERRUN ?
YES
P
NO
CALL KEYBIN TO CHECK ON KEYBOARD INPUT.
WAS ANYTHING ENTERED ?
NO
YES
CALL GETTOK TO RETURN THE RUN-TIME COMMAND.
BLANK COMMAND ?
YES
L
NO
CALL GETCMD TO IDENTIFY THE COMMAND.
G

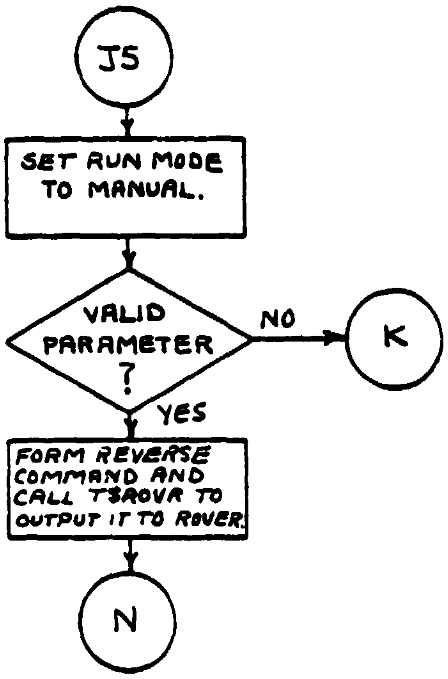

G
LEGAL COMMAND ?
NO
YES
IS A PARAMETER EXPECTED ?
NO
YES
CALL GETTOK TO RETURN THE PARAMETER.
PARAMETER MISSING ?
YES
NO
CALL CNVPAR TO CONVERT PARM. TO A NUMERIC.
ERROR ON CONVERSION ?
YES
NO
K
H

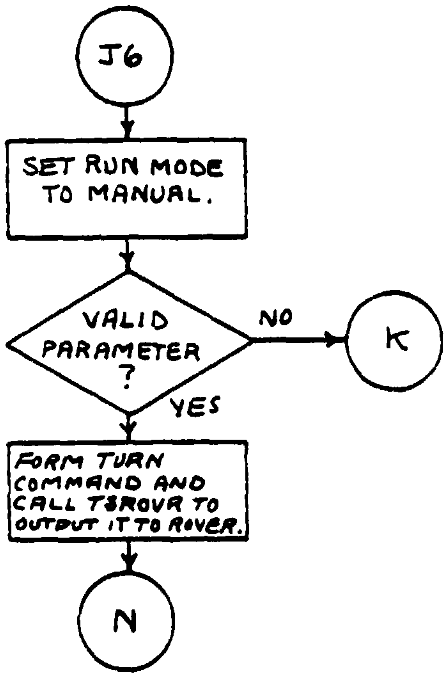

H
COMMAND
1
2
3
4
5
6
J1
J2
J3
J4
J5
J6

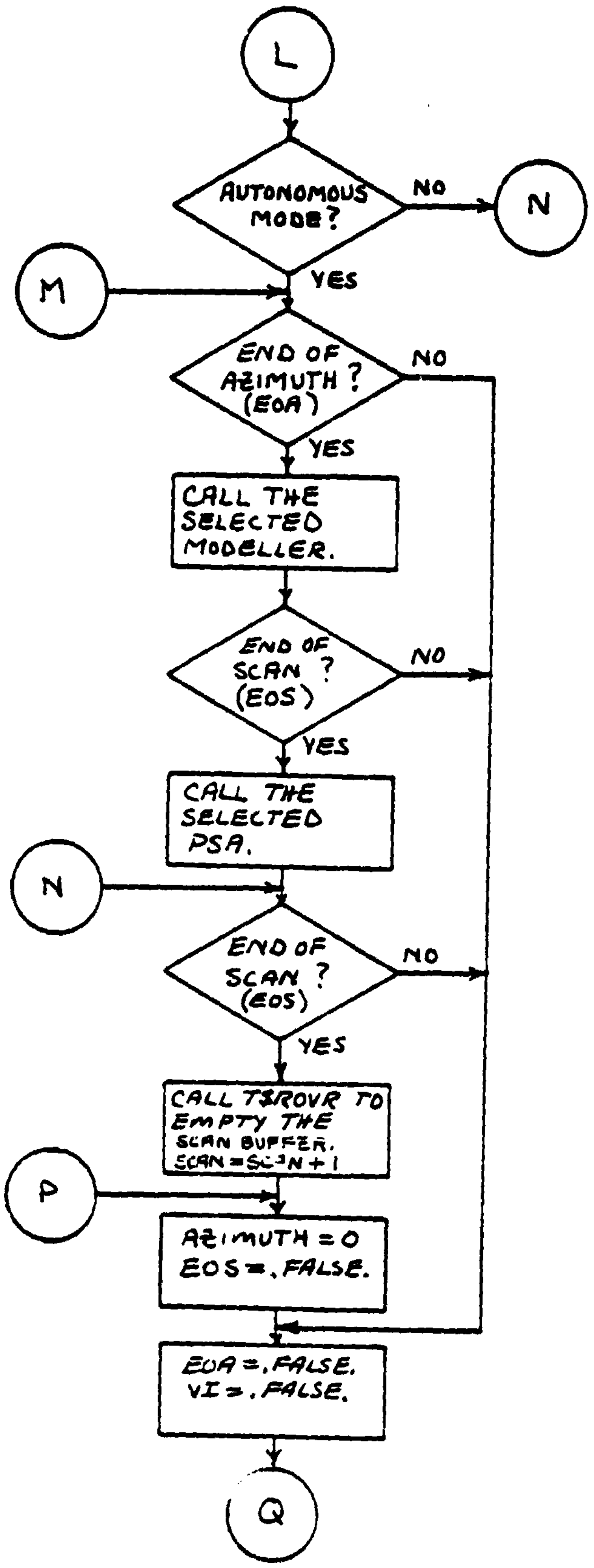

K
OUTPUT ERROR MARKER FOR COMMAND LINE
L


J1
START AUTONOMOUS RUN:
RNAUTO = .TRUE.
M

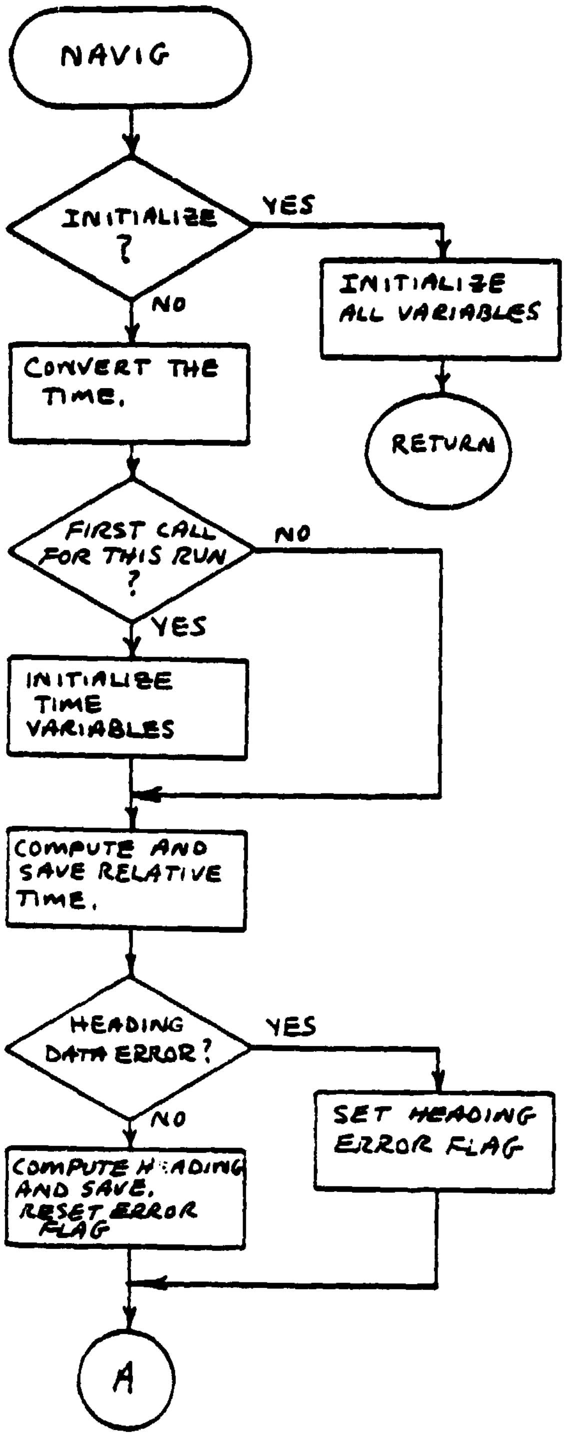

J2
CALL T$ROVR TO OUTPUT STOP COMMAND TO ROVER. SET RUN MODE TO MANUAL.
N
J3
CALL T$ROVR TO HALT THE RUN. CLOSE THE POST-PROCESSOR FILE.
B
J4
SET RUN MODE TO MANUAL.
VALID PARAMETER ?
NO
K
YES
FORM FORWARD COMMAND AND CALL T$ROVR TO OUTPUT IT TO ROVER.
N

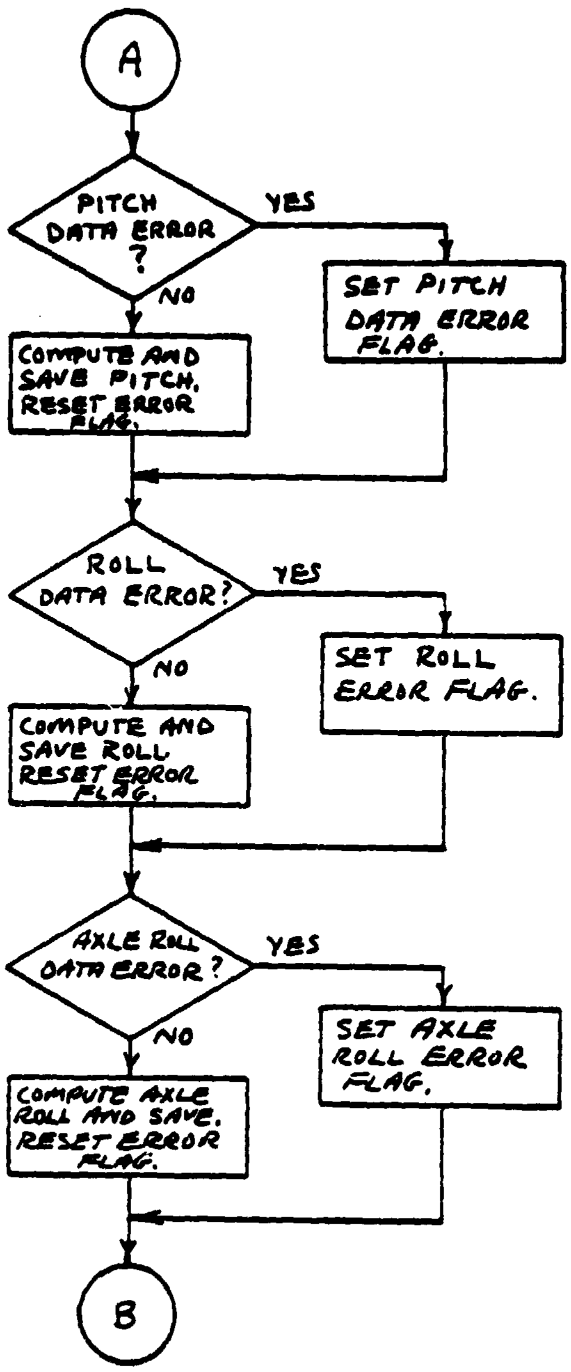

J5
SET RUN MODE TO MANUAL.
VALID PARAMETER ?
NO
K
YES
FORM REVERSE COMMAND AND CALL T$ROVR TO OUTPUT IT TO ROVER.
N

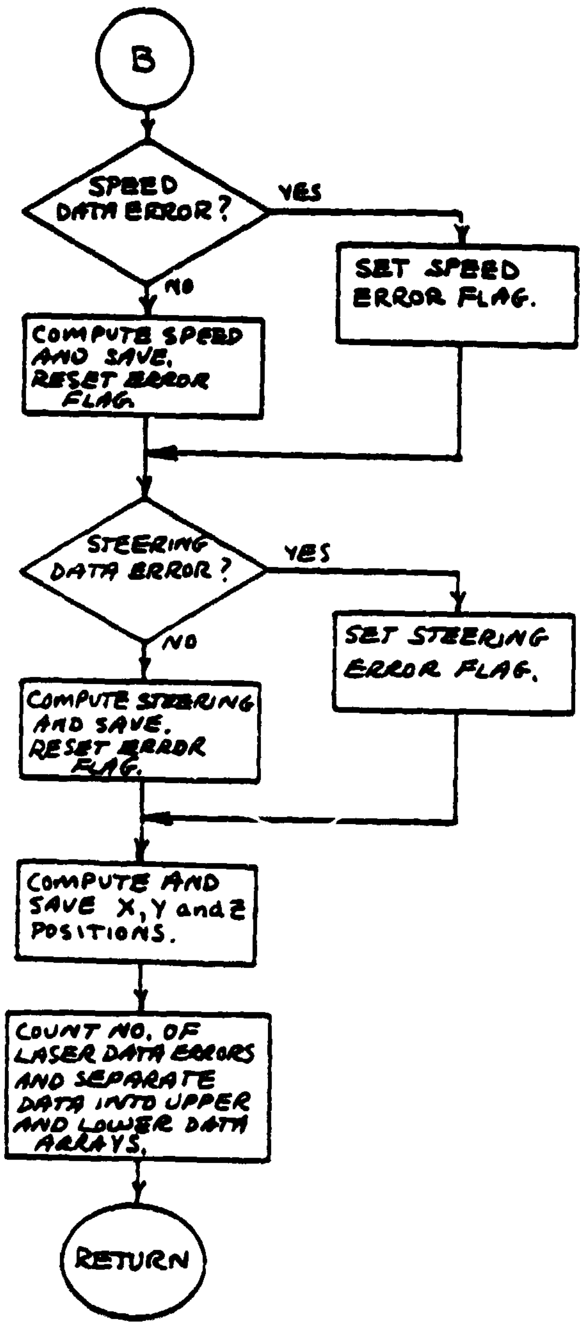

J6
SET RUN MODE TO MANUAL.
VALID PARAMETER ?
NO
K
YES
FORM TURN COMMAND AND CALL T$ROVR TO OUTPUT IT TO ROVER.
N

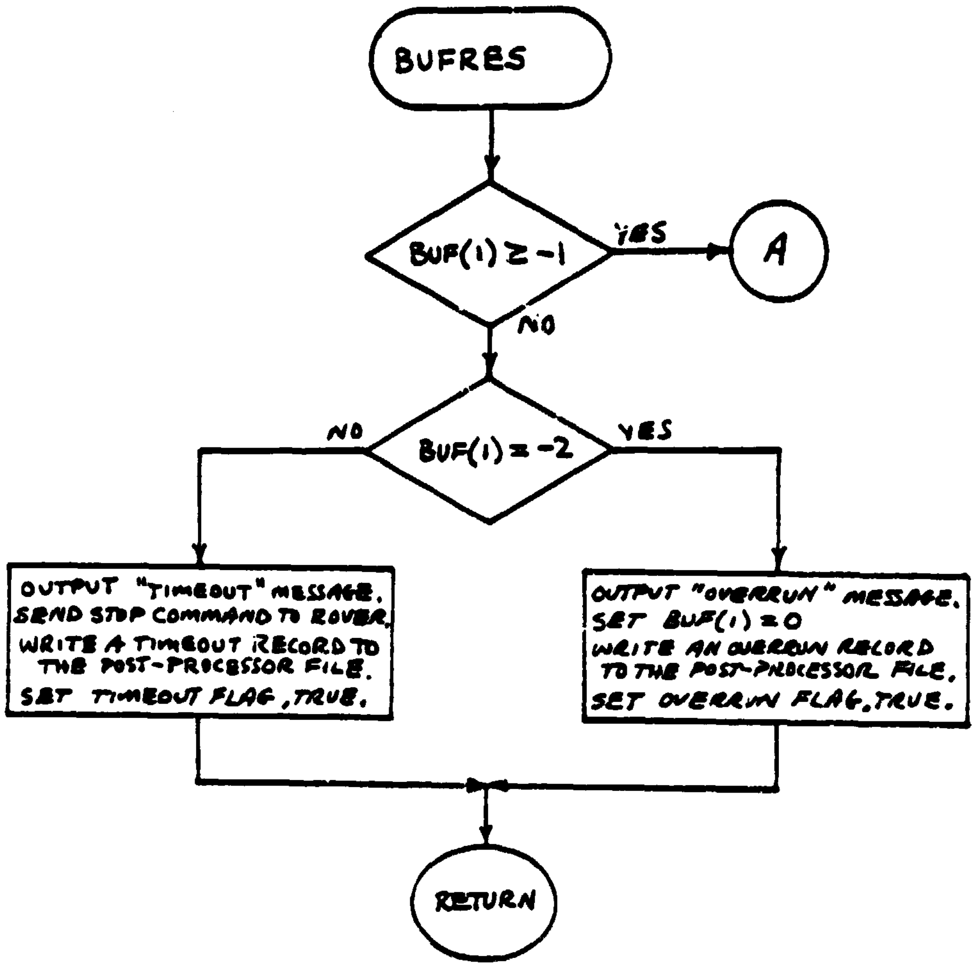


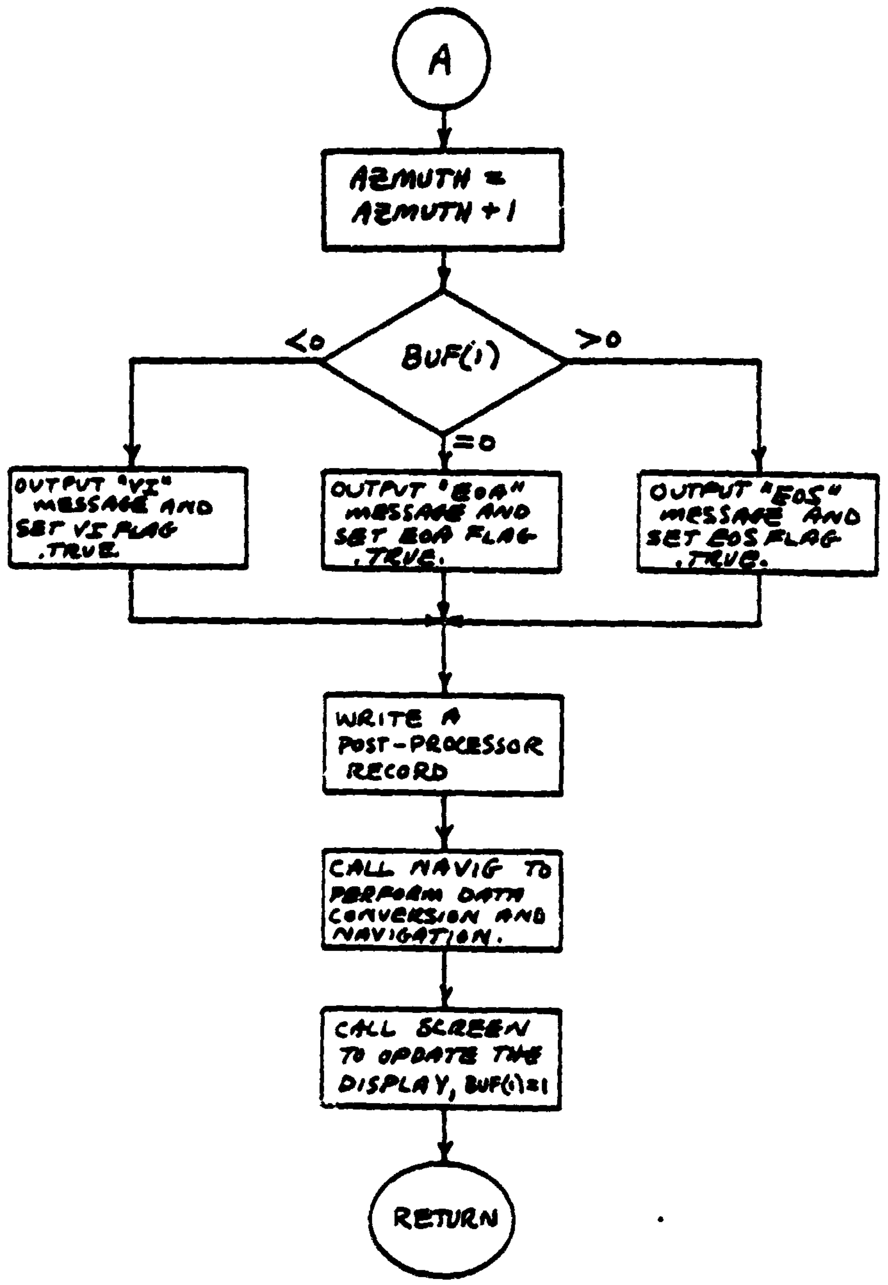
L
AUTONOMOUS MODE?
NO
N
YES
M
END OF AZIMUTH ? (EOA)
NO
YES
CALL THE SELECTED MODELLER.
END OF SCAN ? (EOS)
NO
YES
CALL THE SELECTED PSA.
N
END OF SCAN ? (EOS)
NO
YES
CALL T$ROVR TO EMPTY THE SCAN BUFFER. SCAN = SCAN + 1
P
AZIMUTH = 0 EOS = .FALSE.
EOA = .FALSE. VI = .FALSE.
Q

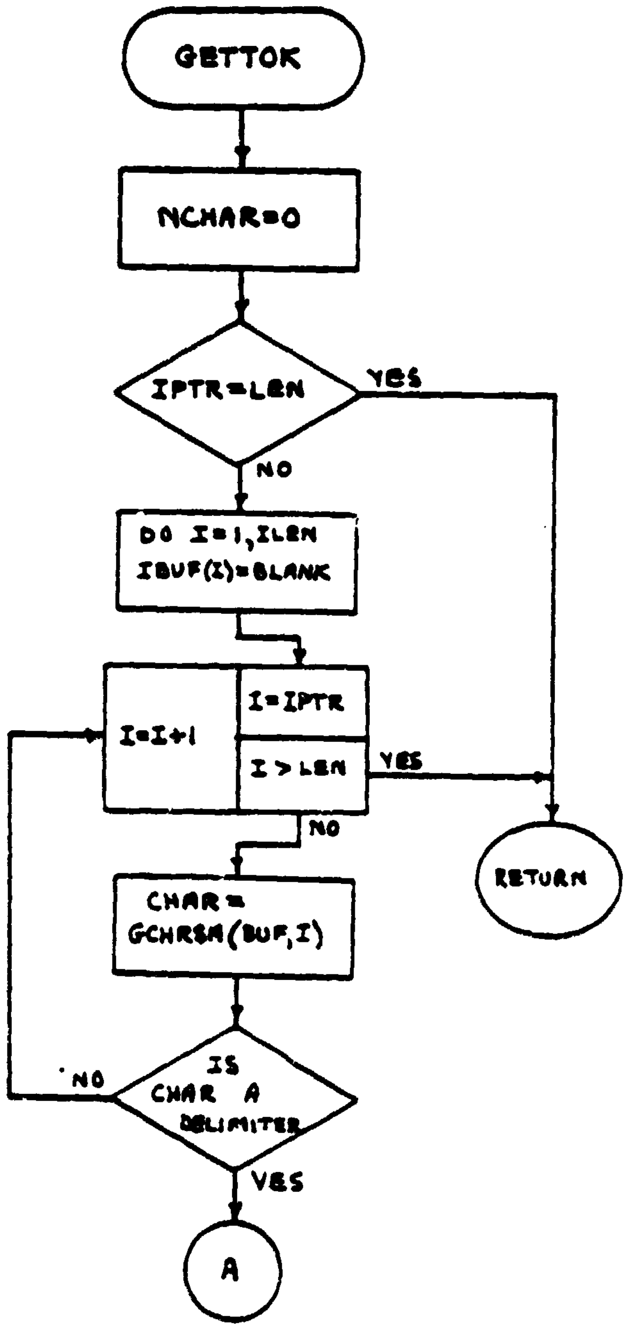

NAVIG
INITIALIZE ?
YES
NO
INITIALIZE ALL VARIABLES
RETURN
CONVERT THE TIME.
FIRST CALL FOR THIS RUN ?
NO
YES
INITIALIZE TIME VARIABLES
COMPUTE AND SAVE RELATIVE TIME.
HEADING DATA ERROR ?
YES
NO
SET HEADING ERROR FLAG
COMPUTE HEADING AND SAVE. RESET ERROR FLAG
A

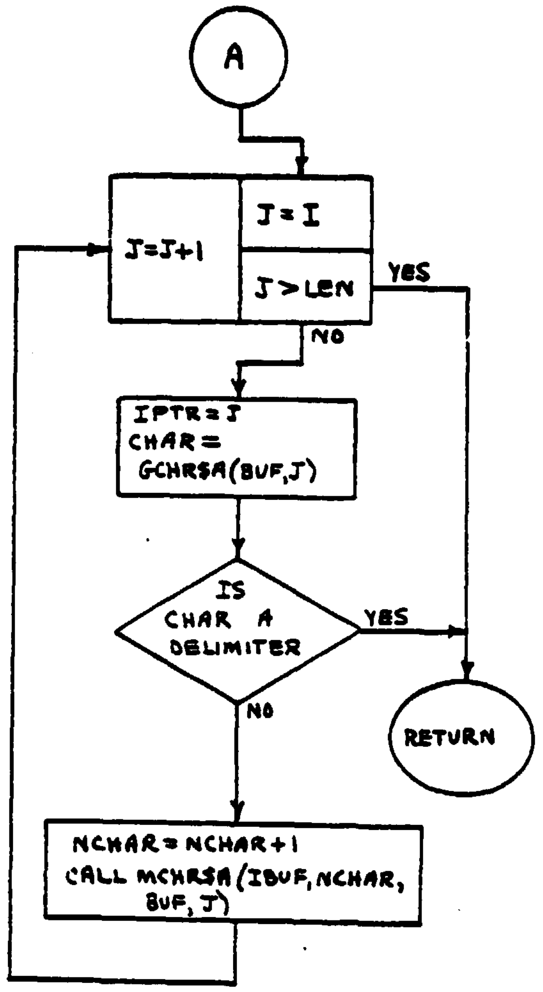

A
PITCH DATA ERROR ?
YES
NO
SET PITCH DATA ERROR FLAG.
COMPUTE AND SAVE PITCH. RESET ERROR FLAG.
ROLL DATA ERROR?
YES
NO
SET ROLL ERROR FLAG.
COMPUTE AND SAVE ROLL RESET ERROR FLAG.
AXLE ROLL DATA ERROR ?
YES
NO
SET AXLE ROLL ERROR FLAG.
COMPUTE AXLE ROLL AND SAVE. RESET ERROR FLAG.
B

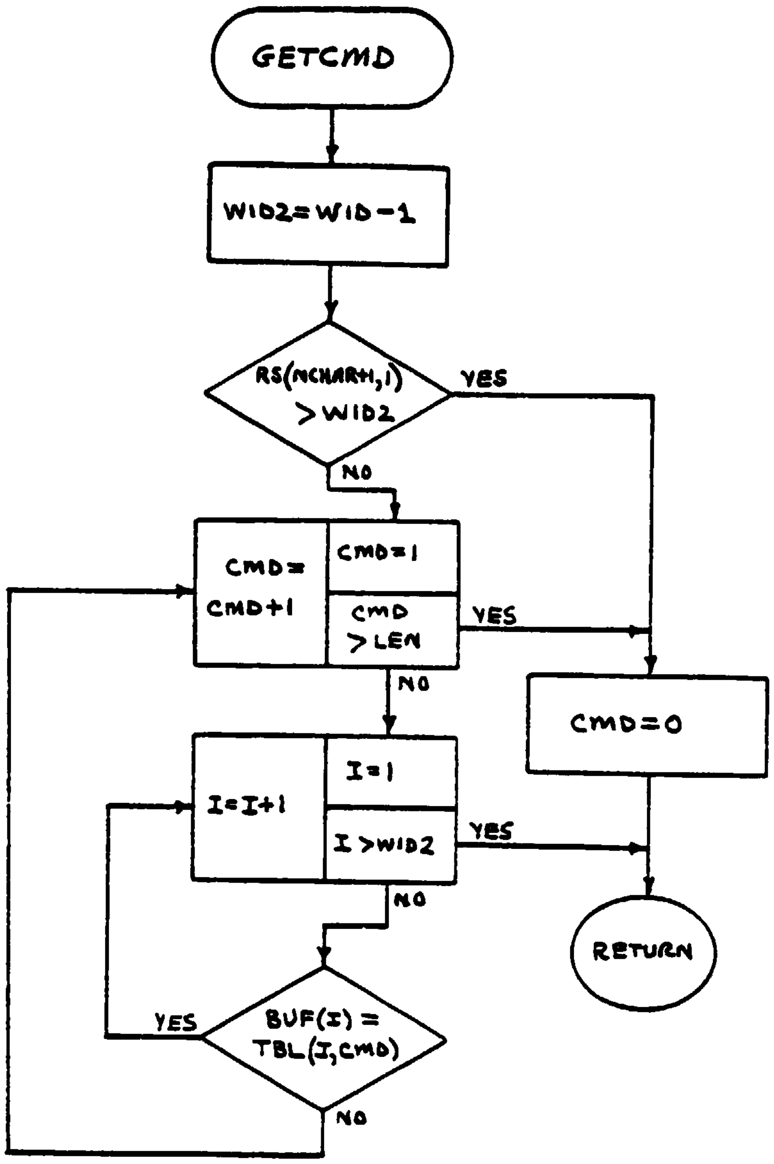

B
SPEED DATA ERROR?
YES
SET SPEED ERROR FLAG.
NO
COMPUTE SPEED AND SAVE. RESET ERROR FLAG.
STEERING DATA ERROR?
YES
SET STEERING ERROR FLAG.
NO
COMPUTE STEERING AND SAVE. RESET ERROR FLAG.
COMPUTE AND SAVE X, Y and Z POSITIONS.
COUNT NO. OF LASER DATA ERRORS AND SEPARATE DATA INTO UPPER AND LOWER DATA ARRAYS.
RETURN

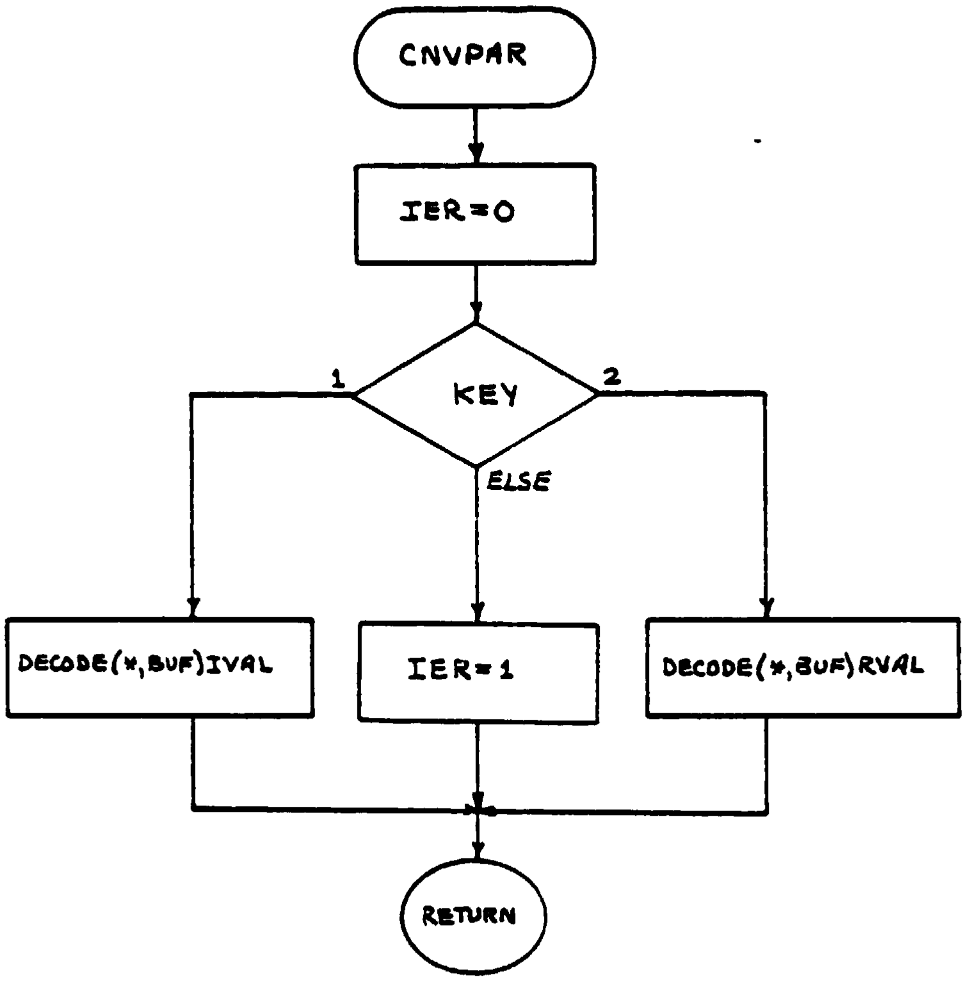

BUFRES
BUF(1) ≥ -1
YES
A
NO
BUF(1) = -2
NO
YES
OUTPUT "TIMEOUT" MESSAGE.
SEND STOP COMMAND TO ROVER.
WRITE A TIMEOUT RECORD TO THE POST-PROCESSOR FILE.
SET TIMEOUT FLAG .TRUE.
OUTPUT "OVERRUN" MESSAGE.
SET BUF(1) = 0
WRITE AN OVERRUN RECORD TO THE POST-PROCESSOR FILE.
SET OVERRUN FLAG .TRUE.
RETURN

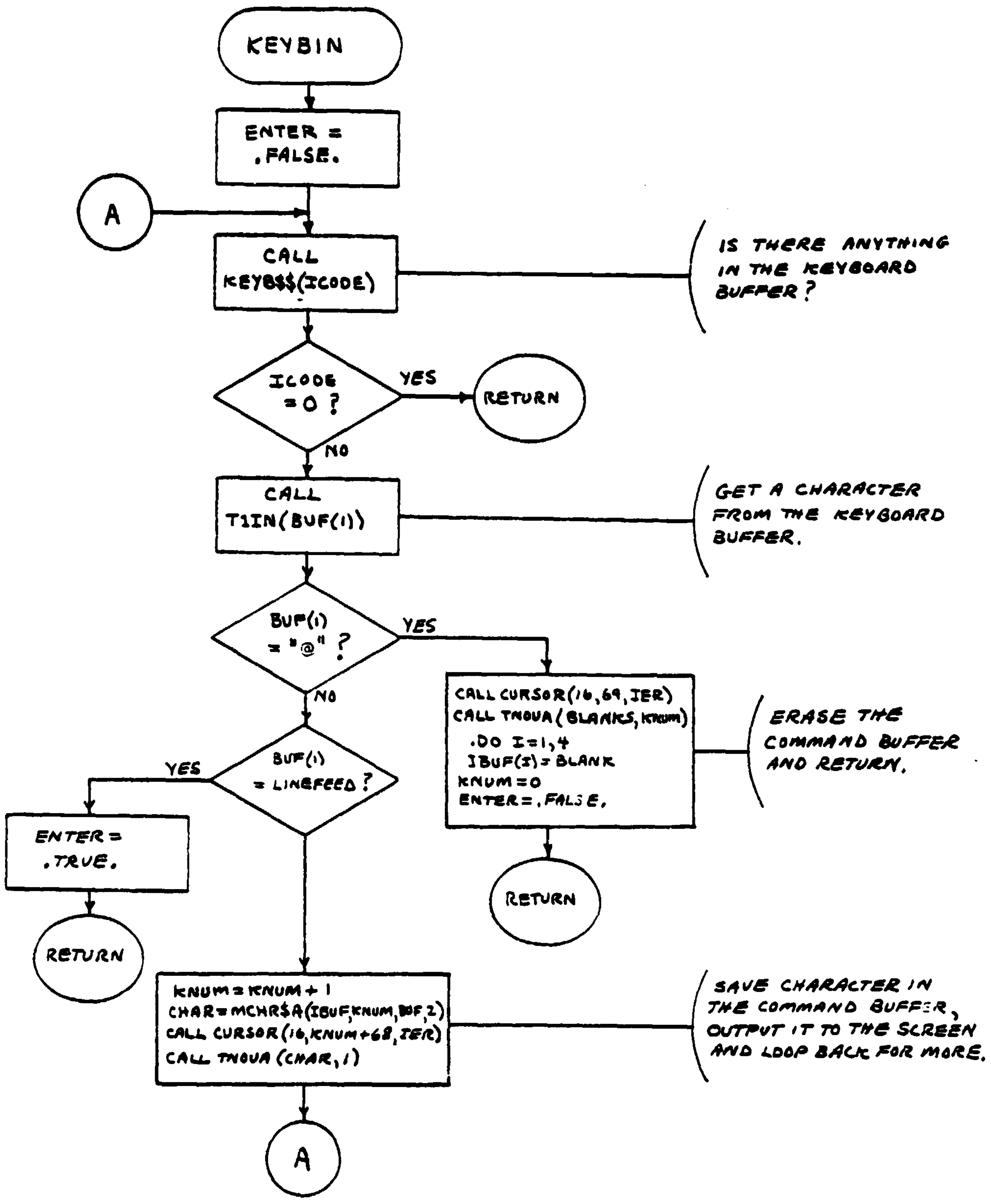

A
AZMUTH = AZMUTH + 1
BUF(I)
<0
=0
>0
OUTPUT "VI" MESSAGE AND SET VI FLAG .TRUE.
OUTPUT "EOA" MESSAGE AND SET EOA FLAG .TRUE.
OUTPUT "EOS" MESSAGE AND SET EOS FLAG .TRUE.
WRITE A POST-PROCESSOR RECORD
CALL NAVIG TO PERFORM DATA CONVERSION AND NAVIGATION.
CALL SCREEN TO UPDATE THE DISPLAY, BUF(I)=1
RETURN

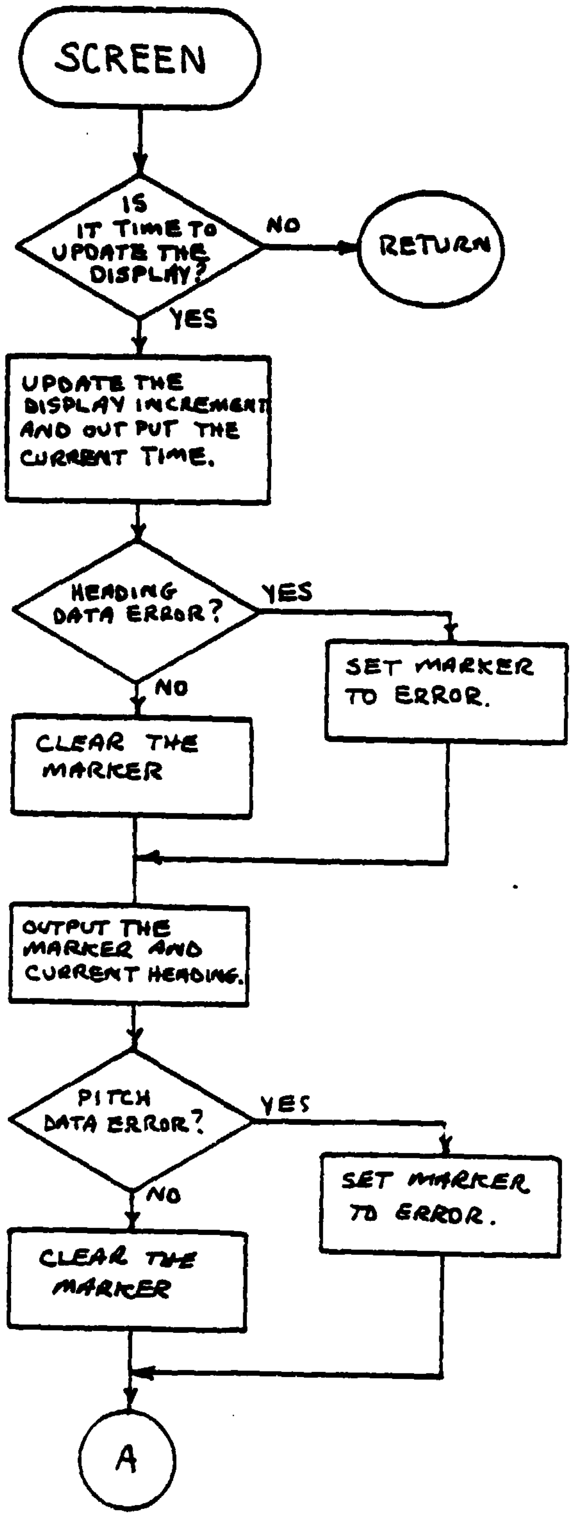

GETTOK
NCHAR=0
IPTR=LEN
YES
NO
DO I=1,ILEN
IBUF(I)=BLANK
I=I+1
I=IPTR
I > LEN
YES
NO
RETURN
CHAR=
GCHRSM(BUF,I)
IS
CHAR A
DELIMITER
NO
YES
A

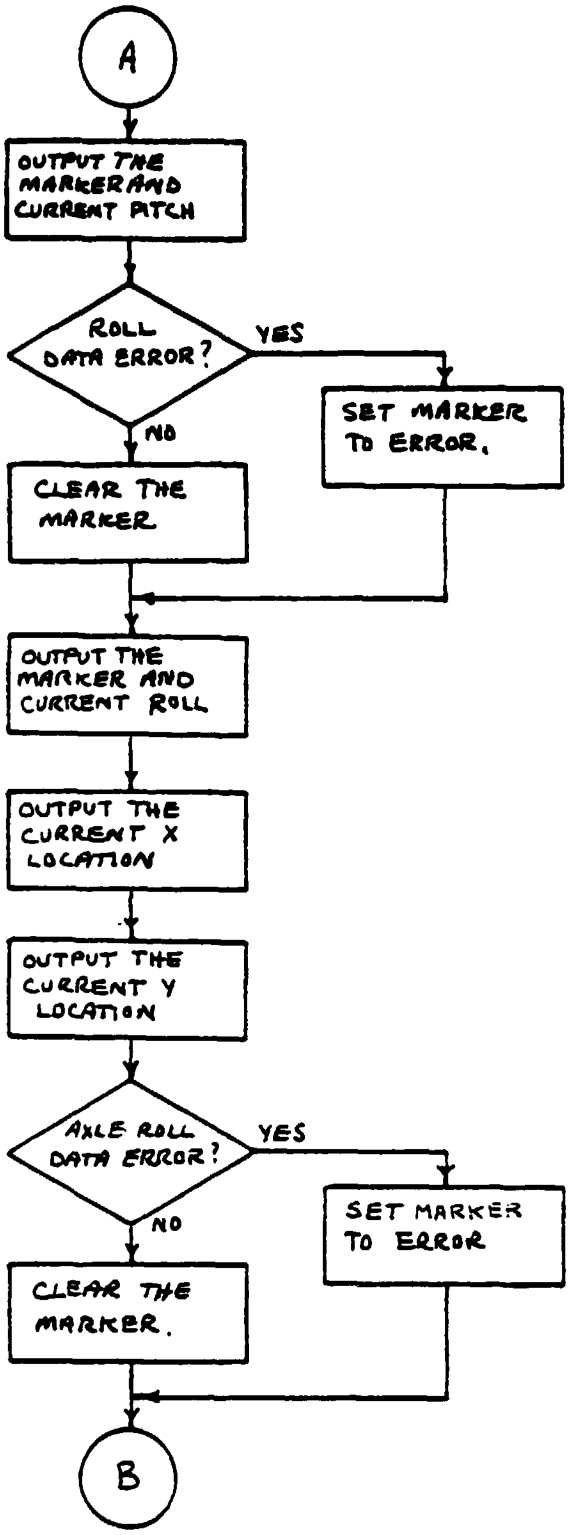

A
J=I
J=J+1
J>LEN
YES
NO
IPTR=J
CHAR=
GCHR$A(BUF,J)
IS
CHAR A
DELIMITER
YES
NO
RETURN
NCHAR=NCHAR+1
CALL MCHR$A(IBUF,NCHAR,
BUF,J)

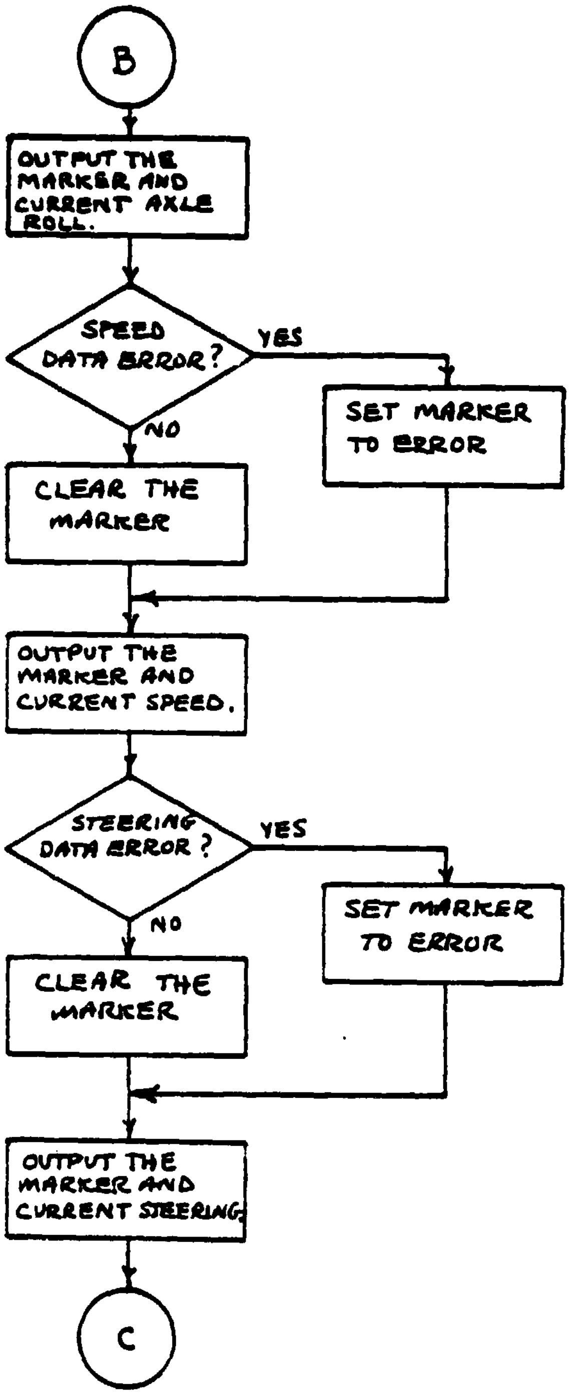

GETCMD
WID2=WID-1
RS(NCHARH,I) >WID2
YES
NO
CMD=CMD+1
CMD=1
CMD >LEN
YES
NO
CMD=0
I=I+1
I=1
I >WID2
YES
NO
RETURN
BUF(I) = TBL(I,CMD)
YES
NO

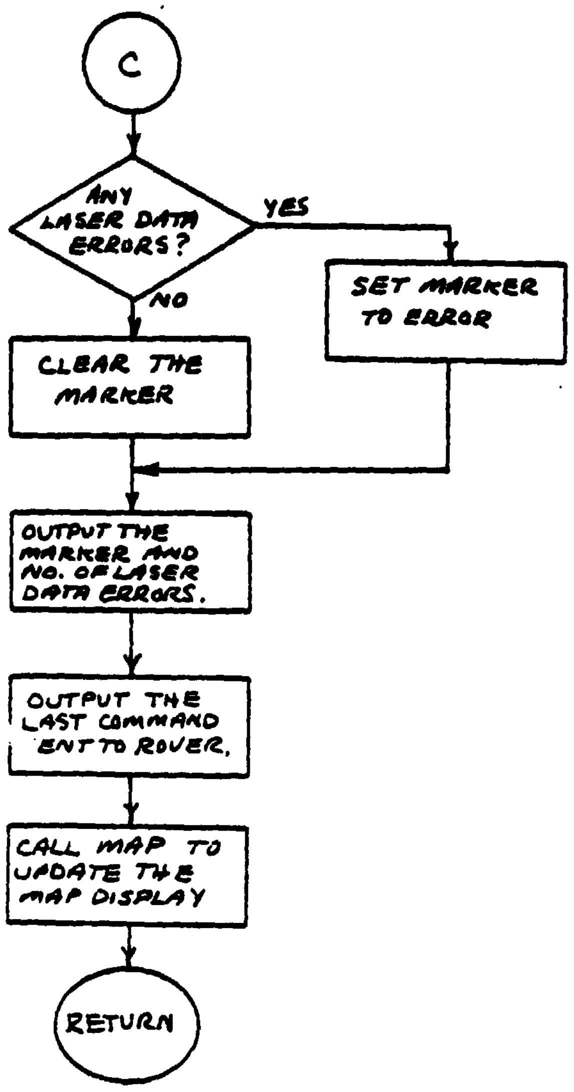

CNVPAR
IER=0
KEY
1
2
ELSE
DECODE(*,BUF)IVAL
IER=1
DECODE(*,BUF)RVAL
RETURN

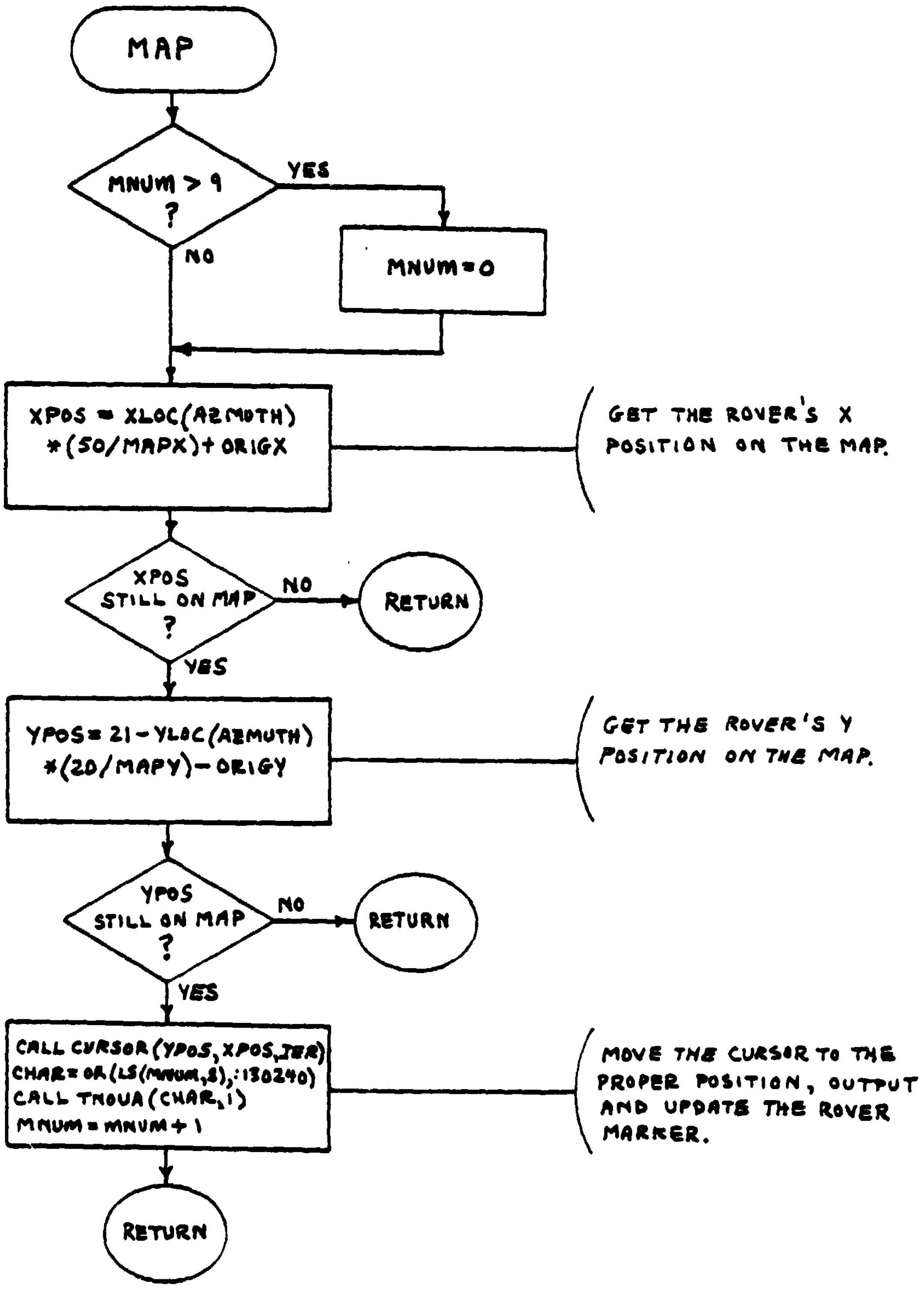

KEYBIN
ENTER = .FALSE.
A
CALL KEYB$$(ICODE)
IS THERE ANYTHING IN THE KEYBOARD BUFFER?
ICODE = 0 ?
YES
RETURN
NO
CALL T1IN(BUF(1))
GET A CHARACTER FROM THE KEYBOARD BUFFER.
BUF(1) = "@" ?
YES
NO
CALL CURSOR(16,69,IER)
CALL TNOUA(BLANKS,KNUM)
.DO I=1,4
IBUF(I)=BLANK
KNUM=0
ENTER=.FALSE.
ERASE THE COMMAND BUFFER AND RETURN.
RETURN
BUF(1) = LINEFEED ?
YES
ENTER= .TRUE.
RETURN
KNUM=KNUM+1
CHAR=MCHR$A(IBUF,KNUM,BUF,2)
CALL CURSOR(16,KNUM+68,IER)
CALL TNOUA(CHAR,1)
SAVE CHARACTER IN THE COMMAND BUFFER, OUTPUT IT TO THE SCREEN AND LOOP BACK FOR MORE.
A

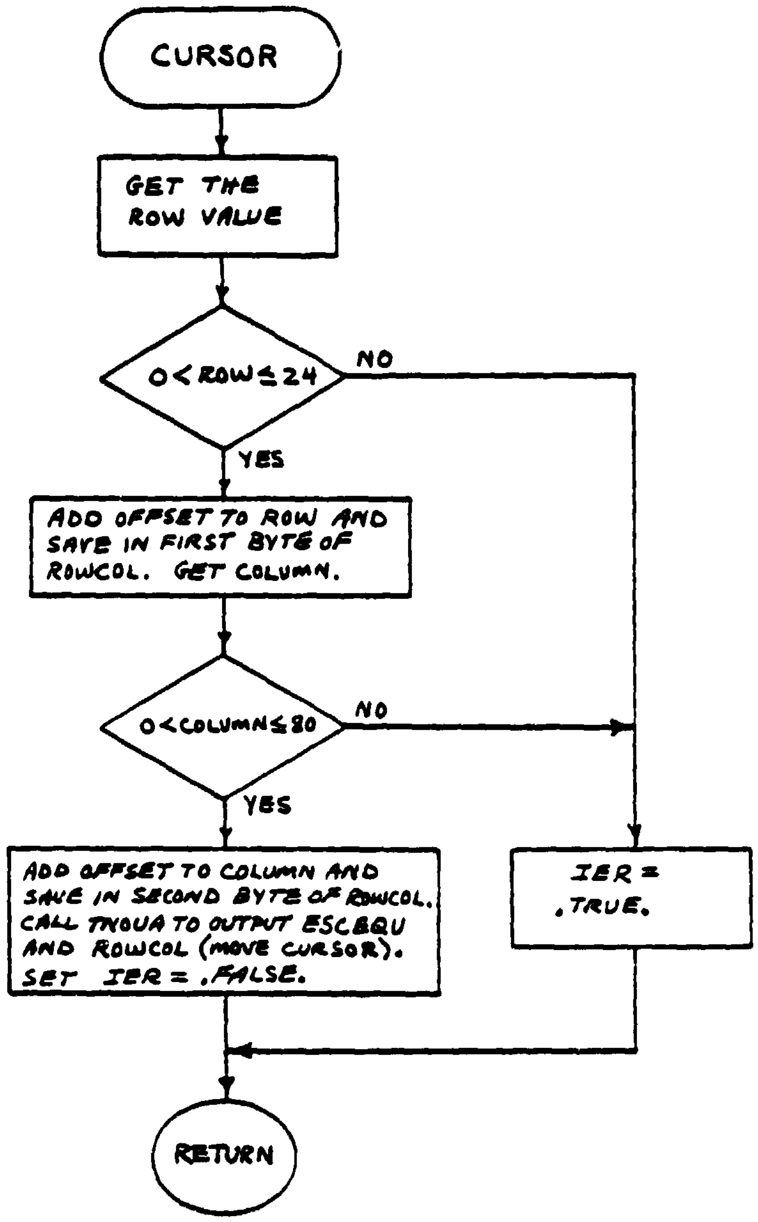

SCREEN
IS IT TIME TO UPDATE THE DISPLAY?
NO
RETURN
YES
UPDATE THE DISPLAY INCREMENT AND OUTPUT THE CURRENT TIME.
HEADING DATA ERROR?
YES
SET MARKER TO ERROR.
NO
CLEAR THE MARKER
OUTPUT THE MARKER AND CURRENT HEADING.
PITCH DATA ERROR?
YES
SET MARKER TO ERROR.
NO
CLEAR THE MARKER
A


A
OUTPUT THE MARKER AND CURRENT PITCH
ROLL DATA ERROR?
YES
SET MARKER TO ERROR.
NO
CLEAR THE MARKER
OUTPUT THE MARKER AND CURRENT ROLL
OUTPUT THE CURRENT X LOCATION
OUTPUT THE CURRENT Y LOCATION
AXLE ROLL DATA ERROR?
YES
SET MARKER TO ERROR
NO
CLEAR THE MARKER.
B


B
OUTPUT THE MARKER AND CURRENT AXLE ROLL.
SPEED DATA ERROR?
YES
SET MARKER TO ERROR
NO
CLEAR THE MARKER
OUTPUT THE MARKER AND CURRENT SPEED.
STEERING DATA ERROR?
YES
SET MARKER TO ERROR
NO
CLEAR THE MARKER
OUTPUT THE MARKER AND CURRENT STEERING.
C


C
ANY LASER DATA ERRORS?
YES
NO
SET MARKER TO ERROR
CLEAR THE MARKER
OUTPUT THE MARKER AND NO. OF LASER DATA ERRORS.
OUTPUT THE LAST COMMAND ENT TO ROVER.
CALL MAP TO UPDATE THE MAP DISPLAY
RETURN


MAP
MNUM > 9 ?
YES
NO
MNUM = 0
XPOS = XLOC(AZMUTH) *(50/MAPX) + ORIGX
GET THE ROVER'S X POSITION ON THE MAP.
XPOS STILL ON MAP ?
NO
RETURN
YES
YPOS = 21 - YLOC(AZMUTH) *(20/MAPY) - ORIGY
GET THE ROVER'S Y POSITION ON THE MAP.
YPOS STILL ON MAP ?
NO
RETURN
YES
CALL CURSOR(YPOS, XPOS, IER)
CHAR = OR(LS(MNUM,8), :130240)
CALL TNOUA(CHAR, 1)
MNUM = MNUM + 1
MOVE THE CURSOR TO THE PROPER POSITION, OUTPUT AND UPDATE THE ROVER MARKER.
RETURN


CURSOR
GET THE ROW VALUE
0<ROW≤24
NO
YES
ADD OFFSET TO ROW AND SAVE IN FIRST BYTE OF ROWCOL. GET COLUMN.
0<COLUMN≤80
NO
YES
ADD OFFSET TO COLUMN AND SAVE IN SECOND BYTE OF ROWCOL. CALL TNOUA TO OUTPUT ESCEQU AND ROWCOL (MOVE CURSOR). SET IER = .FALSE.
IER = .TRUE.
RETURN