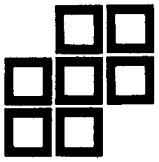
OPUS

USER'S MANUAL



Revision 2.0, January, 1978

I. START UP PROCEDURES

A. GENERAL REQUIREMENTS

OPUS is designed to run on any 8Ø8Ø- or Z8Ø-based computer system, regardless of hardware peripheral configuration. To implement this flexibility, the language must first be tailored to the particular hardware configuration of the system by means of the System Generation Routine. This routine allows the user to declare all drivers necessary to run the terminals, discs, and other devices configured in the system.

The version of OPUS purchased will always contain this System Generation Routine, and prior to actually using OPUS, this routine must be executed. This section is designed to help the user through this routine in order to produce a version of OPUS configured specifically for the hardware in use.

OPUS/ONE CASSETTE VERSION

This version has no capability of accessing disc drives and, hence, no disc commands. Any serial device (including cassette) driver may be defined as a standard device driver (listed in Appendix A.) or may be entered directly in machine code during the generation procedure.

OPUS/ONE DISC VERSION

This version comes with more than one System Generation Routine. The routine to specify depends entirely upon the disc drive to be utilized. A.S.I. provides standard disc routines for specific disc drive systems. If a standard drive does not apply in your case, the "ZZ" version should be specified, allowing the user to enter the machine code of the disc driver directly into OPUS. Note that this "ZZ" version will also contain standard disc drivers. All serial device drivers may be defined as standard device drivers or may be entered in machine code during the generation procedure.

OPUS/TWO DISC VERSION

This version includes all OPUS/ONE commands plus additional commands to facilitate programming. The System Generation Routine provides all standard disc drivers and the capability of entering a user-defined disc driver in machine code. Multiple disc drivers may be specified, allowing the user to access different types of drives simultaneously on-line. All serial device drivers may be defined as standard device drivers or may be entered in machine code during the generation procedure.

In order to load OPUS with the System Generation Routine, it is necessary to have this minimum hardware configuration:

1. 8Ø8Ø or Z8Ø Computer Processor

2. Memory: 16K (OPUS/ONE Cassette or Paper Tape)
   20K (OPUS/ONE Disc), 24K (OPUS/ONE "ZZ")
   24K (OPUS/TWO)

3. Any ASCII Terminal communicating with the computer by means of any I/O Interface.

4. One of the following: Paper Tape Reader
   Cassette
   Hard-Sectored Disc Drive
   Soft-Sectored Disc Drive

B. ENTERING THE LOADER

OPUS, with the System Generation Routine, must be loaded into memory prior to executing. To do this, the user must first put into memory a loader routine which will be used to read OPUS off the medium and store it in memory. The loader routine may be entered into memory by any of the following methods, dependent upon the hardware:

1. Toggled in through the front panel switches, if such switches exist.

2. Through the use of a system monitor if there are no front panel switches.

3. If the system currently has an operating system, it may be possible to enter the loader as a file and retrieve it whenever it is necessary to load OPUS.

4. The loader may be burned into PROM (Programmable Read-Only Memory) located at some high memory location.

The machine code listing and format for possible loaders are listed in Appendix B. at the end of this manuel. The user should carefully scan these loaders to determine if they are compatible with the hardware configuration. If so, the loader may be used as is. If not, the loader must be modified or rewritten according to the specifications given for each loader medium (Appendix B.).

At the end of the loader, the user should insert a port initialization routine to initialize the interface board in order to communicate with the terminal.

C. LOADING OPUS AND THE SYSTEM GENERATION

Before executing the System Generation Routine, the terminal with which the routine is to communicate must be declared. This declaration must be made either before or after OPUS is loaded, depending upon the hardware configuration. Refer to the

following section which applies to your configuration.

1. STANDARD SERIAL I/O INTERFACE

Each version of the generation routine has standard drivers with which to communicate with the System Generation Routine. The table in Appendix A. lists the interface for each driver. In this table, a code (I1, I2, I3) is specified for each possible driver that may be used to communicate with the system generation. This code will be referred to in the following sections.

Even though the interface board to be used may not be one of the standard ones listed, it is possible that the driver may be the same, if the same I.C. chip is used. Refer to Appendix A. for a machine code listing of each of the standard serial device drivers that are used in the generation routine to see if your interface is compatible. If it is not, use the next section on using non-standard interfaces (Section I.C.2., page I-5).

If the terminal is strapped to a port above 63, it is necessary to use the non-standard interface procedure even if the interface board is standard. This is because the switch configuration determining the interface does not have enough bits to allow for a port greater than 63.

a. SYSTEM HAS FRONT PANEL SWITCHES

Go through the following procedure to load OPUS, referring to the table in Appendix A.

1) With the examine switch, examine the first location of the loader.

2) Set the sense switches on the front panel to one of these configurations which determines the I/O interface with which the generation routine is to communicate (the sense switches are the left-hand 8 bits of the address switches--A15 is the most significant, A8 the least significant):

A15	A14	SWITCH CODE
Ø	Ø	I1
Ø	1	I2
1	Ø	I3
1	1	I4

3) Set the remaining sense switches (A8-A13) to the lowest port number (whether data or status channel) at which the interface board is strapped for the terminal.
Examples:

A13	A12	A11	A10	A9	A8	PORT
Ø	Ø	Ø	Ø	Ø	Ø	Ø

A13	A12	A11	A10	A9	A8	PORT
Ø	1	Ø	Ø	Ø	Ø	16 (2Ø Octal)
1	1	Ø	Ø	Ø	Ø	48 (6Ø Octal)

4) Mount the medium upon which OPUS is received into the appropriate device.

PAPER TAPE: Start the reader at the beginning of the leader portion of the tape (null bytes).

CASSETTE : PLAY at the beginning of the cassette. Wait 10 seconds.

DISC : Turn the power on.

5) Push RUN on the front panel of the computer. The System Generation Routine will be loaded into memory and control immediately sent to the routine. If the computer should HALT, a checksum error has occurred, and the loading procedure should be restarted.

b. SYSTEM HAS NO FRONT PANEL

Follow this procedure:

1) Mount the medium upon which OPUS is received into the appropriate device.

PAPER TAPE: Start the reader at the beginning of the leader portion of the tape (null bytes).

CASSETTE : PLAY at the beginning of the cassette. Wait 1Ø seconds.

DISC : Turn the power on.

2) With the system monitor, start execution at the beginning of the loader. Do not let the loader automatically send control to the generation routine (modify if necessary). Return control to the monitor. If the system HALTS before OPUS is loaded, a checksum (bad read) has occurred. Restart the procedure.

3) With the monitor, examine the first three bytes of OPUS:

3Ø3 JUMP instruction (Octal)
L Low address
H High address

4) Examine the location given by H and L above. There should be the

following:

OCTAL	HEX
363	F3
061	31
?	?
?	?
333	DB
377	FF

The fifth and sixth instruction execute an IN from the front panel (port 377/FF). Because there is no front panel, it is necessary to load the Register A with a byte specifying the interface and port.

Bits 7 and 6 (most significant) determine the interface:

BIT 7	BIT 6	SWITCH CODE
Ø	Ø	I1
Ø	1	I2
1	Ø	I3
1	1	I4

Bits Ø-5 specify the lowest port number (whether data or status) for which the I/O board is strapped for communication with the terminal. Examples:

BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT Ø	PORT
Ø	Ø	Ø	Ø	Ø	Ø	Ø
Ø	Ø	1	Ø	Ø	Ø	8
1	1	Ø	Ø	Ø	Ø	48

5) Deposit the following two bytes at the location where there is currently a 333, 377 (Octal) or DB, FF, (Hex):

OCTAL	HEX	
Ø76	3E	
A	A	Where A is the byte determined in Step 4, above.

6) With the monitor, start execution at the first byte of OPUS. The OPUS System Generation Routine should begin execution.

2. NON-STANDARD SERIAL I/O INTERFACE

Proceed through the following steps. If the system has a system monitor, it may be used to examine and change memory. Otherwise, use the front panel switches.

a. Load OPUS with the loader by mounting the paper tape, cassette, or diskette, and executing the loader at the first location. Do not allow the loader to send control to OPUS immediately. Return control to the system monitor, or HALT the machine.

b. An input and output subroutine must be deposited at some location in memory, through which the generation routine can communicate with the terminal. The location should not conflict with the location of OPUS (Ø-16K: OPUS/ONE Cassette; Ø-2ØK: OPUS/ONE Disc; Ø-24K: OPUS/TWO).

If a system monitor is being used, it may be possible to use most or all of the input/output routines that the monitor uses.

These subroutines will be used only temporarily by the initialization routine.

SPECIFICATIONS: INPUT ROUTINE

1) The subroutine should execute a loop, waiting for valid data to come in by checking the status channel.

2) When data is present, execute an IN instruction off the data channel.

3) Execute a RET instruction to return the byte in Register A to the OPUS routine.

4) Do not destroy the contents of the B, C, D, E, H, and L registers.

SPECIFICATIONS: OUTPUT ROUTINE

1) OPUS will send the ASCII byte to be output to the terminal in Register B.

2) The subroutine should execute a loop, waiting for an OK status to output the byte.

3) When it is alright to output the byte, move the contents of Register B to Register A and execute an OUT instruction.

4) Execute a RET instruction to return to the OPUS routine.

5) Do not destroy the contents of the B, C, D, E, H, and L registers.

Note: Refer to Appendix A. for examples of input and output subroutines.

Deposit these subroutines in memory.

c. At the following locations relative to the start of OPUS (normally Ø), deposit the following bytes:

OCTAL	HEX	DATA BYTE
ØØØ Ø1Ø	ØØ Ø8	Low address of input subroutine
ØØØ Ø11	ØØ Ø9	High address of input subroutine
ØØØ Ø12	ØØ ØA	Low address of output subroutine
ØØØ Ø13	ØØ ØB	High address of output subroutine

d. The generation routine must read in a 377/FF from port 255 in order to use the user-defined I/O subroutines. If the system has a front panel, all 8 sense switches must be up (high). If the system does not have a front panel, it is quite likely that there is nothing connected to port 255, in which case a 377/FF will automatically be read in from this port, and nothing else needs to be done. However, if there is some device on this port, or if some byte other than a 377/FF may be read, go through the following:

1) The first instruction at the start of OPUS (location Ø) is a JUMP instruction to the start of the System Initialization Routine. Examine the location to which this JUMP instruction sends control. There should be:

OCTAL	HEX
363	F3
061	31
?	?
?	?
333	DB
377	FF

Replace the 333/DB and 377/FF bytes with the following:

Ø76	3E
377	FF

Instead of reading data off port 255, the Register A is immediately loaded with the 377/FF byte.

e. Start execution of OPUS at the starting location. The System Generation Routine should come up.

D. THE SYSTEM GENERATION ROUTINE

The System Generation Routine requires data from the user to determine buffer sizes, I/O drivers and several other parameters. ALL DATA DISCS MUST BE FORMATTED WITHIN THIS ROUTINE. There is a default value for all buffer sizes which are assumed if the user simply hits RETURN to the prompt. Whether a default or user-entered value, the buffer sizes declared will become default sizes under normal OPUS operation. Some may be changed during OPUS with the SET command. Refer to the glossary to get an understanding of the meanings of these buffers. If the user does not have a feel for any of the sizes, it is recommended that the default value be indicated.

All underlining refers to computer prompts.

***** OPUS SYSTEM GENERATION *****

COPYRIGHT (C) 1977 BY ADMINISTRATIVE SYSTEMS, INCORPORATED (A.S.I.)

OCTAL OR HEX NOTATION:

Enter an "O" for Octal or an "H" for Hex notation. All numbers entered in the following generation routine must be entered in this Octal or Hex notation. The generation routine will also output all numbers in this notation.

IMPORTANT !!! No numbers must be entered as decimal numbers. This applies to the routine only, not normal OPUS execution.

INCLUDE MATH FUNCTIONS?

"Y": The following functions will be included in the OPUS command set:

SIN
COS
TAN
ATN
SQR
LOG
EXP
↑ (power)

"N": The above functions will not be included in the command set. Approximately 1400 bytes will be saved if they are not included.

Default: "Y"

TOP MEMORY PAGE:

Enter the number of pages of contiguous memory which OPUS will utilize. A page is 256 bytes (or the 8 most significant bits of a 16-bit address). For example, if there is 32K of memory, enter 200 (Octal) or 80 (Hex).

Default: 2ØØ in Octal or 80 in Hex (128 decimal)

--- BUFFER MAXIMUMS ---

DIGIT PRECISION:

Enter the default value for the maximum number of digits to which precision will be carried out under arithmetic operations.

Default: 1Ø in Octal or 8 Hex (8 decimal)

BYTES- VARIABLE NAME TABLE:

Enter the number of bytes determining the buffer size of the table holding all variable names used within a program.

Default: 226 in Octal or 96 in Hex (150 decimal)

BYTES- OPERAND TABLE:

Enter the number of bytes determining the buffer size of the table temporarily holding the addresses of all operands needed by one operator.

Default: 55 in Octal or 2D in Hex (45 decimal)

BYTES- CONSTANT TABLE:

Enter the number of bytes determining the buffer size of the table temporarily holding the values of expression calculations.

Default: 4ØØ in Octal or in 1ØØ Hex (256 decimal)

BYTES- INPUT LINE:

Enter the maximum number of characters allowed on one line of input.

Default: 12Ø in Octal or 5Ø in Hex (8Ø decimal)

The following prompt refers to disc versions only:

MAXIMUM DATA FILE BUFFERS:

Enter the maximum number of data files which may be accessed simultaneously during a program.

Default: 3

--- SERIAL I/O DRIVERS ---

Each peripheral device on the system other than the discs must be assigned a logical device number defining the device for OPUS operation. These

device number assignments are arbitrary, although it is recommended that Device 1 be the master terminal. There is no limit to the number of devices allowed. During this procedure, the user will also define drivers for each of the devices, whether standard or non-standard peripherals. All devices entered below must be serial access peripherals only.

DEVICE ØØ1:

INTERFACE: Enter one of the following:

(1) For a standard interface, enter the mnemonic listed with the appropriate interface in Appendix A.

(2) For a non-standard interface, enter "?".

(1) If a standard interface is declared:

LOW PORT#:

Enter the lowest port number on the device as it is strapped in the interface board.

INTERRUPTS ALLOWED?

Enter "Y" (yes) if Control C is to be enabled for input to the device under OPUS operation, breaking the current program. This is a software interrupt. No hardware interrupts are used in standalone OPUS. Normally this should be entered for all input-output terminals.

Enter "N" (no) if the device may not interrupt any operation. This should be entered for any device where Control C is to be ignored or treated as any other data, normally cassettes, paper tape punches, etc.

Default: "Y"

NULL BYTES:

Enter the number of null bytes (ØØØ) that should be generated by OPUS after every carriage return is executed in normal operation. This is used for delay control only. Normally this is not necessary and the default specified.

Default: Ø

The generation routine will print the locations of the input routine, the output routine, and the interrupt routine (if specified) as they are entered in as standard drivers.

INPUT ROUTINE: HHH LLL (High address, Low address)
OUTPUT ROUTINE: HHH LLL
INTERRUPT ROUTINE: HHH LLL

(2) If a non-standard ("?") device is indicated:

INTERRUPTS ALLOWED?

Enter "Y" (yes) if Control C is to be enabled for input to the device under OPUS operation, breaking the current program. This is a software interrupt. No hardware interrupts are used in standalone OPUS. Normally this should be entered for all input-output terminals.

Enter "N" (no) if the device may not interrupt any operation. This should be entered for any device where Control C is to be ignored or treated as any other data, normally cassettes, paper tape punches, etc.

Default: "Y"

NULL BYTES:

Enter the number of null bytes (ØØØ) that should be generated by OPUS after every carriage return is executed in normal operation. This is used for delay control only. Normally this is not necessary and the default specified.

Default: Ø

The user must enter input, output, and interrupt machine code subroutines for the device.

The generation routine will prompt the user for each routine by printing out the address in memory for each byte entered in the subroutine. The user must enter the instruction bytes (in Octal/-Hex) after each address prompt.

INPUT ROUTINE:

HHH LLL

Enter the machine code instruction.
HHH: High 8-bit (page) address of the location
LLL: Low 8-bit address of the location

The generation routine will continue the address prompt, incrementing the address by 1 each time, until the user hits carriage return to the prompt, designating an end to the subroutine.

OUTPUT ROUTINE:

HHH LLL

Likewise, enter the output subroutine.

INTERRUPT ROUTINE: (if specified above)

HHH LLL

Likewise, enter the interrupt routine.

The following gives specifications for the three subroutines that are to be entered above. If it is still not clear exactly how to write one of these routines, it is recommended that the user refer to Appendix A. for the listings of the standard drivers for examples. The format is identical.

SPECIFICATIONS: INPUT SUBROUTINE

1) The subroutine must execute a loop, checking for a valid data byte coming in by monitoring the status channel.

2) Execute an IN instruction off the data channel to retrieve the byte.

3) Return the ASCII data byte in Register A. If parity is present it will be ignored by OPUS.

4) Terminate the subroutine with a RET instruction.

5) Do not destroy the values of the B, C, D, E, H, and L Registers.

SPECIFICATIONS: OUTPUT SUBROUTINE

1) OPUS will send an ASCII byte to output to the device in Register B. There will be no parity.

2) The subroutine should execute a loop, monitoring the status channel to determine when it is okay to output the byte.

3) Move the byte in Register B to Register A and execute an OUT instruction to the data channel.

4) Terminate the subroutine with a RET instruction.

5) Do not destroy the values of the B, C, D, E, H, and L Registers.

SPECIFICATIONS: INTERRUPT SUBROUTINE

1) The subroutine should execute an IN off the status channel and determine whether or not a data byte is forthcoming.

2) If data is on the receive line, make sure the carry bit is set, and execute a RET instruction to return to OPUS.

3) If there is no data on the receive line, make sure the carry bit is clear (Ø), and execute a RET instruction to return to OPUS.

4) If data is present, DO NOT execute an IN instruction from the data channel. After OPUS calls this subroutine, if the carry bit is returned high, it will then call the input routine to receive the data byte.

5) Do not destroy the values of Registers B, C, D, E, H, and L.

Note that all three routines do not have to be entered for every device. The input routine should be entered only if data may be received from the device. The output routine should be entered only if data may be sent to the device. The interrupt routine should be entered only if the user wants a Control C received from the device to be able to interrupt program execution. Normally this would only apply to input/output terminals. OPUS utilizes only software interrupts by means of this interrupt routine. All hardware interrupts are disabled.

The initialization routine will repeat the above device definition routine. Incrementing the device number each time, until the user hits carriage return to "INTERFACE:".

--- PORT INITIALIZATION ROUTINES ---

The user may specify routines that will initialize any I/O integrated circuit chips necessary to allow device communication through the port. This routine will normally determine clock rate, start/stop bits, parity, etc. Refer to the documentation on the interface board to determine what the routine must be for your device configuration.

The following prompt will be repeated indefinitely until a carriage return is struck. Normally, for each device declared in the above I/O definition, an initialization routine should be specified here. OPUS will execute all port initialization routines prior to printing the header message.

INTERFACE:

Enter one of the following:

(1) For a standard interface, enter the mnemonic listed with the appropriate interface in Appendix A. This mnemonic will be the same as that given above in the I/O definition.

The generation routine will print the memory location of the initialization routine as it is inserted within OPUS:

INITIALIZATION ROUTINE: HHH LLL (High address, Low address)

(2) For a non-standard interface, enter "?".

The user must enter the machine code of the initialization routine.

INITIALIZATION ROUTINE:

HHH LLL

Enter the machine code instruction.

HHH: High 8-bit (page) memory location
LLL: Low 8-bit memory location

SPECIFICATIONS:

1. The subroutine must output as many bytes to the status port as needed to declare clock rate, start/stop bits, and any other hardware configuration.

2. Terminate the subroutine with a RET instruction.

3. Do not destroy the values of the C, D, and E Registers.

****

The following applies to OPUS/ONE Disc Version only (not "ZZ"):

--- DISC NUMBER DEFINITION ---

NUMBER OF DISC DRIVES:

Enter the total number of drives using the standard driver on the system.

Default: 2

--- DISC FORMAT ---

ALL DISCS TO BE USED UNDER OPUS FOR PROGRAM AND FILE STORAGE MUST BE FORMATTED IN THIS ROUTINE.

DISC NUMBER:

Enter the number of the disc drive (Ø-?) in which an OPUS data disc is to be formatted. If there are no more to be formatted, hit carriage return.

FORMAT ENTIRE DISKETTE?

Enter "Y" if entire diskette is to be formatted. This must be specified if Altair disc drives are being implemented or if the user wishes to make sure that all sectors on the diskette are good. The routine will write all zeros in each sector of the diskette. If a bad sector is detected, a disc failure will occur.

Enter "N" if the drive is not an Altair and the user is not concerned with condition of the diskette. The routine will write all zeros in just the first track of the diskette. This is sufficient for normal OPUS operation.

IDENTIFYING TAG:

Enter a tag (a label of any ASCII characters, not exceeding 7 characters in length) which will identify the diskette. This tag may be specified in OPUS

disc operations.

INSERT DISKETTE IN DRIVE:

Insert the diskette in the above-specified drive and hit carriage return. The diskette will be formatted.

The format routine will return to "DISC NUMBER:" to format as many diskettes as needed.

****

The following applies to OPUS/TWO and to OPUS/ONE "ZZ" versions only:

--- DISC DRIVER DEFINITION ---

Multiple drivers for disc drives may be entered. This section defines the sector read and sector write subroutines for each different drive that may be running on the system. Multiple discs (defined in the next section) may access the same driver routines. Driver numbers are automatically assigned to each driver routine to be used in the disc definition section. These numbers are temporary only and have no relationship to the disc numbers used in normal OPUS operation.

The following prompts will be repeated continuously until the user hits carriage return to "INTERFACE:".

DRIVER N (N starts with 1 and increments sequentially)

INTERFACE:

Enter one of the following:

(1) Enter a mnemonic of a standard driver listed in Appendix A.

LOW PORT#:

Enter the lowest port number of the driver (each disc usually has a series of port numbers that it will use).

The driver will be inserted in OPUS and the location of the sector read and write subroutines printed:

SECTOR READ ROUTINE: HHH LLL

SECTOR WRITE ROUTINE: HHH LLL

(2) Enter "?" to specify a non-standard driver.

A sector read subroutine and a sector write subroutine must be entered in machine code.

SECTOR READ ROUTINE:

This subroutine must select the required drive, locate the heads on the correct track and sector, and read the proper number of data bytes into the given buffer area.

HHH LLL

Enter the subroutine in machine code instructions. Terminate with a carriage return.

HHH: High 8-bit address of memory location
LLL: Low 8-bit address of memory location

SECTOR WRITE ROUTINE:

This subroutine must select the required drive, locate the heads on the correct track and sector, and write the proper number of bytes from the buffer area onto the disc.

HHH LLL

Likewise, enter the sector write subroutine

SPECIFICATIONS FOR BOTH READ AND WRITE SUBROUTINES:

1. Upon entry to the subroutine, these registers will have the following values:

   C Disc drive number Ø-?
   L Sector number Ø through S-1 (S= number of sectors per track)
   H Track number Ø through T-1 (T= tracks per disc)
   DE Double register pair containing the address of the buffer into which data must be read or from which data must be written

2. The routine must return to OPUS one of the following values in Register A:

   Ø Successful read operation

   1 Multiple attempts were needed to read the sector, but a successful read did finally occur. The routine itself must be responsible for rereading a sector. OPUS will print out a "DISC RETRY" message, giving the disc number track and sector of the retry.

   2 The routine was unable to successfully read the sector. A "DISC FAILURE" message will occur, giving the disc number, track and sector. Control will be transferred back to command mode.

3. Do not destroy the values of Registers D, E, H, and L.

4. Terminate the subroutine with a RET instruction.

5. The subroutine is responsible for all CRC/Checksum analysis. Retries must be attempted within the routine itself.

6. The subroutine must keep track of the current head location on each drive, if necessary.

7. Many drives specify sector numbers as 1-S instead of Ø-(S-1). Be sure to increment the sector number received by OPUS to handle this situation.

--- DISC NUMBER DEFINITION ---

A disc number must be assigned to each drive in the system, defining the appropriate driver and the physical specifications of the drive. The disc number will be used to reference the drive under OPUS operation.

The following prompts will be issued continuously until the user hits carriage return to "DISC DRIVER:".

DISC N (Where N is the disc number, starting with Ø)

DISC DRIVER:

Enter one of the driver numbers assigned above to define the driver routines for this disc.

DRIVE NUMBER:

Enter the physical drive number (Ø-?) of the disc as it resides in the set of drives.

BYTES/SECTOR:

Enter the number of bytes that will be read/written as data in the sector.

Normal: 2ØØ in Octal or 8Ø in Hex (128 decimal)

SECTORS/TRACK:

Enter the number of sectors on each track.

Normal:

Hard-sectored: 4Ø in Octal or 2Ø in Hex (32 decimal)
Soft-sectored: 32 in Octal or 1A in Hex (26 decimal)

TRACKS/DISC:

Enter the number of tracks on the disc.

Normal: 115 in Octal or 4D in Hex (77 decimal)

--- DISC FORMAT ---

ALL DISCS TO BE USED UNDER OPUS FOR PROGRAM AND FILE STORAGE MUST BE FORMATTED IN THIS ROUTINE.

DISC NUMBER:

Enter the number of the disc drive (Ø-?) in which an OPUS data disc is to be formatted. If there are no more to be formatted, hit carriage return.

FORMAT ENTIRE DISKETTE?

Enter "Y" if entire diskette is to be formatted. This must be specified if Altair disc drives are being implemented or if the user wishes to make sure that all sectors on the diskette are good. The routine will write all zeros in each sector of the diskette. If a bad sector is detected, a disc failure will occur.

Enter "N" if the drive is not an Altair and the user is not concerned with condition of the diskette. The routine will write all zeros in just the first track of the diskette. This is sufficient for normal OPUS operation.

IDENTIFYING TAG:

Enter a tag (a label of any ASCII characters, not exceeding 7 characters in length) which will identify the diskette. This tag may be specified in OPUS Disc operations.

INSERT DISKETTE IN DRIVE:

Insert the diskette in the above specified drive and hit carriage return. The diskette will be formatted.

The format routine will return to "DISC NUMBER:" to format as may diskettes as needed.

****

All versions of OPUS:

--- DUMP OPUS ---

ENTER DEVICE NUMBER:

Enter one of the following:

(1) To dump OPUS to a serial device (cassette, paper tape, etc.), enter the device number (1-?) as defined in the I/O definition.

PREPARE DEVICE

For paper tape, turn the punch on. For cassette, insert a blank cassette, push RECORD, and wait 10 seconds.

Hit carriage return. OPUS will be dumped to the specified medium.

(2) To dump OPUS to a diskette, enter "D" (does not apply with OPUS/ONE cassette version).

DISC NUMBER:

Enter the disc number (as defined in the disc number definition section) on which OPUS is to be dumped.

INSERT DISKETTE IN DRIVE:

Insert a blank diskette in the specified drive, and hit carriage return. OPUS will be dumped on this disc.

--- FINISHED ---

The generation routine will go into a loop. The system may be halted. The user's version of OPUS may at any time be loaded by following the "BRINGING UP OPUS" procedure on the following pages.

NOTE:

It is possible to test the tailored version of OPUS to make sure that all parameters and drivers have been entered correctly without loading this version. After OPUS is dumped, it resides in memory exactly as it will when loaded. Thus simply start execution at the first location (normally Ø). OPUS should come up.

IMPORTANT !

If OPUS does not behave correctly after being initialized, carefully check all parameters and drivers entered in the generation routine. It is most likely that something was entered incorrectly.

E. BRINGING UP INITIALIZED OPUS

To bring up the initialized version of OPUS, the following steps must be implemented:

1. Put in Loader

   If the initialized version of OPUS was dumped to the same medium from which OPUS with the System Generation Routine was loaded, the same loader should be used. If the medium was different, a corresponding loader must be put into memory. Refer to the loader procedure in Section I. B. to determine how to put the loader in memory; the procedure in this step is identical to that in this Section.

2. Define Device & Load OPUS

   Front Panel

   Before executing the loader, the sense switches of the front panel must be set to determine the device upon which OPUS is to come up. Assuming "Ø" means switch down and "1" means switch up, the following table gives the switch patterns corresponding to device numbers defined in the System Generation Routine:

A15	A14	A13	A12	A11	A10	A9	A8	Device Number
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1
Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	2
Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	3
Ø	Ø	Ø	Ø	Ø	Ø	1	1	4

   No Front Panel

   Using the loader, load OPUS into memory. Normally OPUS will first execute an IN instruction from port 255 (377 in Octal, FF in Hex). If either a Ø or 255 (377/FF) is read in, control will be sent to Device 1. If any other byte is possible, OPUS must be patched to determine the device. Change the following locations (shown in both Octal and Hex):

Location	Old Byte	New Byte	Location	Old Byte	New Byte
ØØØ 1Ø1	333	Ø76	ØØ 41	DB	3E
ØØØ 1Ø2	377	ØØØ	ØØ 42	FF	ØØ

   By making this patch, OPUS will always come up on Device 1. The second byte may be changed to give a different device number according to the above switch table.

3. OPUS Execution

Start execution of OPUS at the first location. OPUS should print:

```
* * * * *  OPUS/???  REV ???  * * * * *

COPYRIGHT (C) 1977 BY ADMINISTRATIVE SYSTEMS INCORPORATED (A.S.I.)

DAY?      Enter the day of the month (0-31)

MONTH?    Enter the month (1-12)

YEAR?     Enter the last two digits of the year (ØØ-99)

FINE
```

OPUS is up and running!!! Good Luck!!!

II. INTRODUCTION TO THE OPUS LANGUAGE

A. BACKGROUND

A.S.I. was incorporated in the fall of 1975 with the intent of forming a small computer company utilizing the new technology of microcomputers. As is probably the case with hundreds of microcomputer hobbyists, the potential of these small machines seemed unlimited and extremely exciting. At first, A.S.I. was primarily interested in the end-user product or application, in particular, business applications. It was our intention to utilize a BASIC language that was to be on the market shortly. However, it became apparent that it would be several months before we would see this language. Finally, we made the decision to go ahead and develop our own high-level language, not with the intent of selling the language itself, but as a tool for our end-user applications. So OPUS came into being.

OPUS started out as a very BASIC-like language. However, as development proceeded, the question, "Why does this language have to follow the rules of BASIC?", increasingly kept producing the answer, "It doesn't". As applications were our end goal, it seemed only logical to build a language that would make these applications easy and efficient to produce. BASIC is a very rigid and structured language, with little room for programming flexibility. For an advanced programmer, the limitations become quickly apparent and coding becomes tedious. We felt we could develop a tool that overcame these limitations, producing a language that provided a never-ending challenge.

Not because structured programming was the 'in' thing to do, but more because a structured language provides more logical code with less debugging time, a block structure (similar in many ways to ALGOL) was built into OPUS. This allows programs to be written in a very logical flow, with little jumping all over the place, and almost no need for the questionable GOTO statement. Automatic internal conversions from numbers to strings and vice versa frees the programmer from that common programming headache, and almost unlimited number precision (up to 55 digits) allows the programmer to worry about more important things than double or triple precision routines. But that is what this manual is all about -- to explain all the commands and the ins and outs of OPUS.

OPUS was developed primarily as an interpreting language as opposed to a true compiler. An interpreter scans lines of program code and operates according to the instruction. A compiler takes the program code and compiles it into machine code instructions, essentially inserting the operational subroutines from the language as needed. The machine code is then directly executable. There is a great difference of opinion as to whether interpreters or compilers are the best way to go. Compilers may improve speed somewhat and are useful because a program may be executed without the operating system being present. On the other hand, interpreters allow more flexibility in terms of editing and debugging and

usually take less memory to operate. OPUS attempts to utilize the best of two worlds by compiling program code into an object code format, allowing a very quick interpretation of the code during execution. Overhead in interpretation is cut to a minimum. In addition, there is a run-only subset of OPUS available (FORTE) which is dedicated to executing finished programs in a minimal amount of memory.

We eventually decided to go ahead and market OPUS. Feeling strongly that we had produced a high-level language that matched the excitement and potential of the microcomputer industry, OPUS was put into preliminary production in March of 1977 to run on any 8Ø8Ø- or Z8Ø-based microcomputer with any combination of peripheral devices. Since that period, we have continued to improve upon the language, in fact producing a couple more versions with extended capabilities. We feel strongly that the end result both challenges and rewards the programmer.

This language and this manual are not for all programmers or potential programmers. First, it is assumed that the person using this manual has had some experience with computer languages. The manual is not a programming course. It explains all the commands and operations of the OPUS language, without going into all the detail of exactly what a programming language is, or in fact, what a computer is. Secondly, the very flexibility of OPUS, a plus factor for advanced programmers, may cause headaches and frustration for the programmer with little experience with block-structured languages. There is no denying that a language with rigid and simple rules will be easier for the beginning programmer. However our feeling is that with patience, imagination, and a certain amount of perseverance, the richness and potential of OPUS will produce more rewards than gray hairs. Thirdly, OPUS is probably not the language for scientific applications requiring a rigid speed factor in numerical calculations. Because no hardware multiply/divide boards are used, and variable precision is implemented, OPUS will break no records for number-crunching speeds, for business applications, games, etc., this variable precision capability more than makes up for a little lost speed, which will not be at all apparent in most instances.

B. LANGUAGE CONSTRUCTION

This section is intended to give the programmer the basic knowledge needed to understand and operate OPUS. It should be read very carefully. The more understanding gained from this section will eliminate many problems when programming is actually undertaken. Most terms and procedures necessary for a basic understanding of OPUS are thoroughly explained in this section.

Command Mode

Upon bringing up OPUS and responding to the date prompts (day, month, year), the system will print FINE and go into what is called "command mode". In this mode, the programmer may either type commands directly to OPUS or edit a program. If the first character entered in the line is not a number, OPUS assumes the programmer has entered a command. After a carriage return is struck, the line of statements will automatically be compiled and directly executed. Following execution, OPUS will always respond with FINE, indicating completion. As many operations as needed may be typed on a single line. All statements in OPUS may be executed in command mode, although some will have little meaning in such an environment. Essentially one line of statements executed in command mode is much like writing a one-line program and running it. Under normal operation, command mode is used primarily to initiate some process, such as loading or running a program, or checking the values of variables during debugging.

Examples of statements in command mode:

```
PRINT "TESTING";
TESTING

FINE
LOOP I,1 to 5; PRINT I*I; NEXT;
1
4
9
16
25

FINE
GET "PROG"

FINE
LIST

   10  REM "THIS IS AN EXAMPLE OF A PROGRAM THAT DOES NOTHING"
   20  END;

COM

FINE
RUN

FINE
```

Note that the maximum length of a line entered is determined in the System Generation Routine; the default value is 80 characters.

Program Editing

If a number is first entered on a line in command mode, OPUS assumes that it is being entered as a line in a program, to add, change, or delete. All lines in a program must be preceded by a line number -- any integer from 0 to 65535. These line numbers are used only for editing programs; when the program is compiled and executed, the line numbers are deleted and are of no use. GOTOs and other statements must refer to labels within the program and not to line numbers.

When a line number is entered in command mode, OPUS will first check to see whether program code has been entered after the line number.

If no code is entered after the line number, OPUS assumes that the line specified by the number is to be deleted from the program currently in memory. The line, if it exists, will be deleted. If it does not exist, it will be ignored. Example:

```
LIST

  10  REM "EXAMPLE"
  20  PRINT "BOO";
  30  END;

FINE
20
LIST

  10  REM "EXAMPLE"
  30  END;

FINE
```

If code is entered after the line number, OPUS will scan the program in memory for a matching line number. If this line number is already in the program, OPUS will replace that line in the program with the new line just entered. Example:

```
LIST

  10  REM "EXAMPLE"
  20  PRINT "BOO";
  30  END;

FINE
20 PRINT "HI";
LIST

  10  REM "EXAMPLE"
  20  PRINT "HI";
  30  END;
```

If any portion of a line is to be changed, the entire line must be re-entered in this fashion.

If the line number entered in command mode does not exist in the current program in memory, OPUS will add the line to the program. The location of this line is determined by the numerical value of the line number. The starting line in a program will always have the lowest line number, and the last line, the highest line number. Any integer from Ø to 65535 may be used to place the line of code in the correct location in the program. It may be necessary to use the RENumber command to renumber a program if the programmer finds it necessary to add a line between two lines with a line number difference of one. Example of new line entered:

```
10 REM "I AM ENTERING LINE 10"
5 PRINT "THIS IS LINE 5";
100 PRINT "THIS IS LINE 100";
LIST

   5  PRINT "THIS IS LINE 5";
  10  REM "I AM ENTERING LINE 10"
 100  PRINT "THIS IS LINE 100";

FINE
```

Note that after entering a line number in command mode, OPUS will not respond with FINE but will simply drop to the next line. This indicates that no direct command has been executed, just that the program has been modified. OPUS does not scan the code entered after a line number for syntax and no errors are possible at this point.

Special Characters

There are several special characters that apply to editing and controlling features in OPUS. These are mostly control characters. That is, the user must push the control (CTRL) key down and while holding it, press the character key. Control characters are listed below with a ↑ preceding the character.

Carriage Return (CR): After entering any line of input (in command mode or within a program), the return key must be hit. This tells OPUS that the line is complete and to continue on.

↑H : May be used when entering a line and a mistake is made. The last character entered will be deleted and the cursor will move backwards one character. Multiple ↑Hs may be entered to delete more than one character.

Underline : Deletes the previous character exactly as ↑H does, but instead of the cursor backing up, OPUS will generate an underline character.

↑X : If the user is entering a line and decides that the entire line needs to be re-entered, ↑X may be struck. A backlash (\) will be printed and the cursor will drop to the next line. The entire line should be re-entered. Note that ↑X must be used before a carriage return is hit to terminate the line.

↑C : If this character is hit, OPUS will return to command mode and print FINE. It may be used at any time during program or statement operation to terminate the process.

↑S : Hitting this character will suppress all output by temporarily suspending the operation. This is useful when listing programs or generating other output to a CRT. The list or output may be halted briefly to allow the user to scan the contents.

↑Q : This character will release the ↑S function. That is, when the user wishes output to continue after a ↑S has been hit, the ↑Q must be struck.

Source Programs

Programs entered with line numbers are called source programs. The characters entered in the line will first be stored in memory in ASCII format; ASCII code is simply a way of expressing characters in binary code that the computer will recognize. Upon hitting carriage return, OPUS will immediately compact this ASCII line into source code which is significantly shorter in content, and is not straight ASCII code, as was entered. Commands are stored as a one-byte number, variables pretty much the same, numbers in floating point format, strings with a string identifier. The source program entered in this manner is not directly executable. It may only be listed and edited in this form. To run the program, it must be first compiled (COM statement) into object code format.

Compilation

Compilation is the process by which source code is turned into executable object code. The order of the source code is modified to a form more easily interpreted by the computer, commonly know as Postfix or Reverse Polish notation. All operands (constants or variables) are stored prior to the command itself. Operations are normally entered in Prefix notation, the operand following the command, or Infix notation, in which the operator is imbedded between the operands. Actually, all OPUS code may be entered in Postfix notation, but common usage dictates that commands precede the operands. For example:

```
PRINT "HI";
HI

FINE
"HI" PRINT;
HI

FINE
```

The result is the same because compilation from source to object results in identical object code in both instances.

The major reason for storing program code in this Postfix notation is that there is a very simple algorithm to execute such code. Overhead in interpreting code to be executed is cut to a minimum.

An important factor to remember when source programs are compiled to object programs is the idea of priority. Each statement in OPUS is assigned a priority number (listed in Appendix D). Commands with a higher priority will be compiled and executed before those with a lower priority. The user is, no doubt, familiar with this concept in arithmetic expressions. Most programming languages assign priorities to arithmetic operators similar to the following:

Operation		Priority #	
Exponentiation	↑	3	(Highest Priority)
Division	/	2	
Multiplication	*	2	
Subtraction	-	1	
Addition	+	1	(Lowest Priority)

Therefore, an expression such as:

```
A+B/C↑2
```

would first execute C↑2, divide B by the result and finally add A.

```
5*4+2-1
```

would multiply 5 times 4 (2Ø), add 2 (22), and subtract 1 (21).

In OPUS, all statements are handled in this fashion. Priority numbers with a higher number indicating a higher priority are given to each statement. Note that many statements may have the same priority; in which case, the first statement encountered (scanning left to right) will be executed first.

Parentheses are also very important in the compilation process. They may be used to force an operation of lower priority to be executed before one of higher priority. Taking our example above, if one entered the expression as:

```
5*(4+2)-1
```

4 would first be added to 2 (6), multiplied by 5 (3Ø), and finally 1 is subtracted (29).

Parentheses may be nested to any depth. The following:

```
2↑(4*(1+2))
```

Would add 1 and 2 (3), multiply times 4 (12), and raise 2 to this power (4Ø96).

Parentheses may be used with any operation expressions in OPUS, not just arithmetic expressions. This allows statements to be imbedded within other statements. For example, the following operation:

```
If Z>Ø [A="POSITIVE"] ELSE [A="NEGATIVE"];
```

may be written as:

```
A= (IF Z>Ø ["POSITIVE"] ELSE ["NEGATIVE"]);
```

Parentheses do not exist in the object code. It is the compilation process itself that will use the parentheses to determine the order of execution; it will not insert the parentheses as a command.

OPUS will not give an error if the number of left parentheses does not match the number of right parentheses. However, program execution could cause strange results because commands may have been compiled in a manner not intended by the user.

Object Programs

An object program is the source program in compiled form that is directly executable using the RUN command. No line numbers in the source program will appear in the object program. The object program is treated essentially as a one-line program with no differentiation between the lines.

Statements

All instructions or commands in OPUS are normally referred to as statements. It is important to note that every reserved character or word in OPUS has a unique operation, including the statement delimeter ";". Most of the manual is devoted to explaining the operation of each statement available in OPUS.

Operands

Most statements utilize parameters or operands. These operands can be divided into two major groups, constants and variables.

Constants

There are two types of constants, numbers and strings.

Numbers

All numbers in OPUS are stored in normalized floating point format, with a biased exponent and a mantissa stored in BCD (Binary Coded Decimal) notation. Floating point notation is very similar to scientific notation:

```
N.NNN... 1Ø↑X
```

"N.NNN..." is the mantissa and "X" is the exponent (an integer). Both mantissa and exponent may be negative numbers.

The number of digits permitted in the mantissa (precision) is completely under the programmer's control, with a maximum of 55. The default precision is initially set under the System Generation Routine and may be changed at any time with use of the SET command.

If a number may be represented accurately within the precision declared, it will be printed in a normal manner; that is, the decimal point will appear in the correct relative position. However, if the number may not be accurately represented within the precision, it will be printed in its floating point (exponential) format:

N.NNN... E↑X

Where "N.NNN..." is the mantissa, "E" designates exponent, and "X" is that exponent. This is very similar to the scientific notation shown above. "E↑X" is equivalent to "1Ø↑X".

The exponent may have any integer value between -63 and +63, inclusive. Thus, the range of positive numbers in OPUS goes from 1E-63 through 9.999...E63, and negative numbers from -9.999...E63 through -1E-63.

If numerical calculations exceed these limits, OPUS will maintain the maximum exponent limit and print out the following message:

-OVER/UNDERFLOW-

The operation will not be terminated. However, the resulting value will not be accurate because of the underflow or overflow encountered.

At certain times, it is important to know exactly how many bytes a number is going to require for storage. This is particularly important when calculating the amount of data that can fit in a disc file logical record. Note that a number will utilize the minimum number of bytes needed for accurate representation even though this may be less than the number of digits declared as precision.

Each number will require a minimum of 3 bytes. The first byte is a number identifier, giving the number of bytes used by the mantissa. The second byte stores the exponent and the sign of the mantissa. The rest of the bytes are the mantissa. Two digits are stored in each byte in the mantissa. The formula for the total number of bytes is:

Total Bytes = 3 + TRU((# Digits - 1)/2)

where "TRU" indicates truncated integer value. The number of digits refers only to the actual digits in the mantissa, disregarding decimal point and sign. Leading and trailing zeros of a number are not counted

in the number of digits because they are absorbed in the exponent value. Examples:

.334	4 bytes
.000334	4 bytes
33400	4 bytes
1.23456	5 bytes
99	3 bytes
1000000	3 bytes
-.1E50	3 bytes

Strings

A string is a series of alpha-numeric characters; the maximum number of characters in a string is 127. A string may contain any character from ASCII code Ø through code 255.

A string constant within a statement operation must contain the series of characters delimited on each end with a quote ("). Examples:

```
"HORSE CART"
"THIS IS A STING OF LENGTH 29"
"123456789"
"#$%&'(())**"
```

The number of bytes required to store a string will always be the total number of characters in the string plus one. This extra byte is a string identifier giving the number of characters in the string. Quotes do not count as part of the string, and are only used to designate the character series as a string constant.

Note: If a string constant is the last item on a line of code, the last quote is optional. OPUS assumes the string contains all characters to the end of the line.

Portions of strings, or substrings, may be referenced. The user should refer to the "$" command for an explanation (Section VI.).

A null string is a string containing no characters. Its representation as a string constant is:

```
""
```

Null strings may be used at any time, exactly as any other string.

Variables

The user is, no doubt, familar with the concept of variables from any basic math course. A variable can be considered to consist of two parts -- the variable name and the variable value.

The variable name may be any group of upper-case alphabetical characters, the length limited only by the maximum number of characters allowed in an input line. No numbers or other characters are allowed. Also, the variable may not contain any special statement word used by OPUS. Spaces entered between letters of a variable name will be ignored. If memory space is critical, it is recommended that the length of variable names be kept to a minimum. Examples of variable names:

```
X       OK
HOTDOG  OK
Z1      Not OK
TOW     Not OK (Contains TO)
```

The variable value may be thought of as the value assigned to the variable name; this value may be either a number of a string if the variable is a simple variable, or a matrix if the variable is a matrix variable.

Simple Variables

A simple variable will have the value of either a number or a string constant, which may be assigned to that variable at any point during operation. It will always have an initial value of Ø prior to having another value assigned.

OPUS does not differentiate between number variables and string variables. Any variable may hold any value at any time.

Matrix Variables

A variable name is declared to be a matrix variable by using the DIMension statement. The value of this variable is a matrix, which is a series of number and/or string constants, referenced by means of an element number determining the correct dimensioned position. The DIM statement determines the number of dimensions and the number of elements in each dimension. The number of elements in the matrix is the product of the number of elements in each dimension.

A one-dimensioned matrix is an array. For example:

```
DIM M(1Ø);
```

M is now a matrix variable containing a one-dimensioned matrix of 10 elements.

A Tic-Tac-Toe square may be considered to be a two-dimensioned matrix with three elements in each dimension. The rows are the first dimension, and the columns the second.

A cube is a three-dimensioned matrix. To go higher than three dimensions takes a little bit of imagination, but up to 155 dimensions are possible in OPUS.

Once a matrix has been declared in the DIM statement, any element may be referenced or assigned either a number or string value. The format of a matrix element reference is:

variable name (E_1,E_2,...,E_n)

The variable name is the matrix name. E_1 through E_n are the element positions in each dimension, where n is the number of dimensions. The parentheses must be present. In the first line of code in the following example, matrix MX has 2 dimensions, the first with 3 elements, and the second with 2 elements.

```
DIM MX(3,2);

MX(1,1)="ROW 1,COLUMN 1";

LOOP I,1 TO 3; LOOP J,1 TO 2; MX(I,J)=I&","&J; NEXT; NEXT;

PRINT MX;
1,1   1,2   2,1   2,2   3,1   3,2
```

Note: When an entire matrix is referenced, the first element accessed will have the lowest element number of each dimension. Going sequentially, the last dimension will continue to increment by one until the maximum, then the next dimension, etc.

One more example:

```
DIM A(2,3,4);
```

Reading across, the sequential reference would be:

```
A(1,1,1), A(1,1,2), A(1,1,3), A(1,1,4)
A(1,2,1), A(1,2,2), A(1,2,3), A(1,2,4)
A(1,3,1), A(1,3,2), A(1,3,3), A(1,3,4)
A(2,1,1), A(2,1,2), A(2,1,3), A(2,1,4)
A(2,2,1), A(2,2,2), A(2,2,3), A(2,2,4)
A(2,3,1), A(2,3,2), A(2,3,3), A(2,3,4)
```

Once a variable name has been declared to be a matrix variable, it may not be treated as a simple variable. All elements will initially be set to a value of Ø by the DIM statement.

Number ⟷ String Conversion

All values received by OPUS statements may be in either number or string format. Prior to the operation, OPUS will internally convert the value to the correct form required by the statement. Under most circumstances, the programmer need not be concerned with the format of the operand. There are two exceptions:

1. If the operation is comparing two values with a relative operator (such as < or >), it may be necessary to force the values into either both string or both number format (NUM or STR statement). There are specific rules listed in Section X. that determine the manner in which OPUS compares values.

2. When writing data to a disc file, the format of the value is important because the number of bytes required by a string value may be different than that required by its corresponding numerical value.

If a string that does not have a numerical digit as its first character is converted to a number, its value will become Ø. If non-numerical characters follow digits within a string, they will be ignored during the conversion.

Note: Although a value may be converted for an operation, the converted value will be stored only temporarily for use by the operation. If the value belongs to a variable, the format of the variable remains the same.

Examples of number to string conversions:

```
12345              "12345"
-.ØØ1              "-.ØØ1"
5.33E-1Ø           "5.33E-1Ø"

PRINT 1.2 & 34;
1.234
```

Examples of string to number conversions:

```
"33.55"            33.55
"-.ØØ67"           -.ØØ67
"XYZ"              Ø
"8ABC"             8
"*1*2*3"           Ø
"1*2*3"            1

PRINT "35"+"6A";
41

PRINT "DOG"*"HORSE";
Ø
```

Statement Construction

There are essentially three types of statements in OPUS:

1. Unary Operators

   These statements require exactly one operand for execution. This category includes functions and some conditional commands.

General Format: statement expression

2. Binary Operators

These statements require two operands for execution. Normal usage places the operator between the two operands. This category includes arithmetic operators and string concatenation.

General Format: $expression_1$ statement $expression_2$

3. List Operators

These statements require a varying number of operands. This category includes most commands.

General Format: statement $< expression_1, expression_2, . . . >$

A comma is normally used to delimit the expressions. Upon entering an operation, the comma is immediately deleted and therefore, has no executable function.

In the General Format above, the "statement" is the OPUS instruction word. The "expression" may be a string or number constant, a variable, or another operation that will return a value for use by the statement. Note that all statements in OPUS may have operands expressed in this manner. The only thing that OPUS will look at during execution of the statement is the value of the expression, not the syntax.

The carets (< >) designate optional expressions. Whether or not an operand is optional is determined by the specific statement.

This same notation for the format of a statement is used throughout the manual.

Operand Stack

Of primary importance in understanding OPUS is the concept of the operand stack. This stack is a table that holds all constant values, string values, or expression values as they are encountered during execution.

Remember that executable OPUS code is stored in Postfix notation, with the operands preceding the operators. Therefore, upon execution, as OPUS sequentially scans the program, it will encounter constants and variables before it reaches the actual statement. The values of these operands will immediately be pushed onto the operand stack.

When a statement is executed, OPUS will first retrieve from this stack any values that are needed for execution, with the number of values dependent upon the statement. Unary and binary operators will always retrieve the last operand(s) pushed on the stack. List operators will normally use all operands on the stack, the first

on the stack being the first operand retrieved, and so forth.

Most unary and binary operators will not only retrieve operands from the stack but, upon completion, will return the calculated value back to the stack to be used by another operation.

This concept of the operand stack is the reason that operations may be nested within other operations. Examples:

```
Operation:  A = 3*(1+2)
Postfix  :  A 3 1 2 + * =
```

Statement	Operand Stack	Operands Retrieved	Operands Returned
+	A 3 1 2	1 2	3
*	A 3 3	3 3	9
=	A 9	A 9	9
	9		

```
Operation:  PRINT "TEST",(XX = "HOT" & "DOG")
Postfix  :  "TEST" XX "HOT" "DOG" & = PRINT
```

Statement	Operand Stack	Operands Retrieved	Operands Returned
&	"TEST" XX "HOT" "DOG"	"HOT" "DOG"	"HOTDOG"
=	"TEST" XX "HOTDOG"	XX "HOTDOG"	"HOTDOG"
PRINT	"TEST" "HOTDOG"	"TEST" "HOTDOG"	

This may be much more confusing than informative. For simple, straightforward programs, the programmer will probably never need to understand this stack concept. However, as sophistication grows, so will the need to understand this.

The Semicolon

While the operand stack concept is fresh in the reader's mind, the special semicolon statement needs to be explained.

The function of the semicolon is to clear all operands from the stack. Because it has the lowest priority, all code entered before the semicolon will usually be compiled and executed p ior to the semicolon. By executing the semicolon, the programmer is ensured that no operands are left for later operations.

Under normal operations, it is good practice to enter a semicolon after each operation. For example, assume the programmer is in command mode and types:

```
A="HORSE"

FINE
PRINT A
HORSE   HORSE

FINE
```

Why did "HORSE" print twice? The assignment operator (=) returns the value "HORSE" to the operand stack. It is still there, along with the variable A, when the PRINT statement is executed. By using a semicolon after the assignment operation, this problem is corrected:

```
A="HORSE";

FINE
PRINT A
HORSE

FINE
```

Of course, the programmer may have wanted to print "HORSE" twice. The semicolon does not have to be there. As the programmer becomes more familiar with OPUS code, it will become more apparent as to when the semicolon is useful. But, in general, to avoid confusing results like that above, it is good practice to use the semicolon after each operation.

Line Construction

In either command mode or within a program, there is no limit to the number of operations that can be entered on one line, other than the length of the input line. Normally, the semicolon described above will be used as an operation delimiter between operations on the same line.

In command mode, after the user hits carriage return, the entire line of code is compiled and then executed. It is important to note that the operand stack is not touched during compilation of a line in command mode. Therefore, if operands are present on the stack before entering a line, they will be used by the operations in the code.

In a program, line boundaries have virtually no meaning when the program is compiled. Therefore, it is possible to split operations between lines. Example:

```
10   INPUT "WHAT IS YOUR NAME?",NAME; IF LEN NAME>20 [ PRINT
20   "NAME TOO LONG"; END; ] ;
```

Expressions within parentheses are also line independent.

Variable names and constants may not be divided across line boundaries, if an attempt is made to do so, the portion on the first line will be considered one variable or constant, and the portion on the second line a different variable or constant.

Block Construction

It is important in OPUS to understand block construction. A very general descrip-

tion of a block is a section of program code that for some reason must be set apart from the rest of the code. There are several types of blocks in OPUS:

```
[. . .]
LOOP. . .NEXT
WHILE. . .CONT
GOSUB. . .RETURN
CALL. . .RET
```

Unless otherwise specified, the normal usage of the term "block" will refer only to the bracket block first listed above. The other blocks are described under the statement definitions in the manual. The bracket block has primarily two functions:

1. Used in combination with the IF/ELSE or ON statements. These statements will execute code within a block if certain conditions are true, and will otherwise use the brackets to determine which code is to be skipped. Example:

   ```
   IF X>Ø [PRINT "X IS POSITIVE"] ELSE [PRINT "X IS NOT POSITIVE"];
   ```

   If X is greater than zero, the first block will be executed and the second block skipped. If X is less than or equal to zero, the first block will be skipped and the second block executed.

2. The "[" statement (block initiator) will mark a new start to the operand stack. The new start of the stack will become the previous end. Therefore, any operands on the stack before the block is entered will remain there throughout execution of the block even if semicolons are used within the block. The "]" statement (block terminator) will return the address of the stack as it was prior to the block.

   This feature allows for interesting and efficient coding. Operations requiring calculated values may be accomplished within a single operation. Examples:

   ```
   A=([INPUT X; IF NONE [Ø] ELSE [1]]);
   ```

   If a value is entered in the input statement, A is assigned the value of 1, otherwise it is assigned Ø.

   ```
   PRINT (LOOP I,1 TO 5 [I*I] NEXT 1);
   1   4   9   16   2?
   ```

   The value "I*I" will be pushed 5 times onto the operand stack. The print statement will print out all values in the stack.

Parentheses and blocks must not be confused. They have totally separate meanings. Parentheses are used only to specify the order of execution and are not executable statements. Brackets delimit a block of code and are executable statements.

If the code within a block consists of only one statement or one constant, the brackets are not needed. For example:

```
ON X [PRINT 1]\[PRINT 2]\[PRINT 3]
```

may be written as:

```
ON X PRINT 1\2\3
```

or:

```
ON X PRINT [1]\[2]\[3]
```

Matching left and right brackets is very important. OPUS will give no errors if they do not match, but execution will result in strange things. If there is no matching left bracket to a right bracket and the block is skipped, the program will probably terminate as OPUS searches for the matching terminator. If a right bracket is executed without a previous left bracket, a STACK OVERFLOW error and program termination will occur. This function stack holds parameters set up by the block initiator, and, if not present, an error will occur.

Blocks may be nested to any depth within each other. The only limit is memory.

The Colon

The colon is a special statement used to separate operand lists into two sections. Several commands (mostly input/output commands) utilize this statement to distinguish one portion of the list from another. The operands prior to the colon normally have some special meaning and format while those after the colon are a more general list of the values used by the statement.

The actual operation of the colon marks a new start of the operand stack. The operands entered prior to the colon will be in a different section in the stack than those entered after the colon. The statement using the colon will look for the first section in the stack for any special parameters. If there, it will remove them and shift the start of the operand stack back to its initial state. Colons are usually optional. If not present, default values will be assumed. Examples:

```
PRINT 2: "BOO"      Prints "BOO" to Device 2
PRINT 2, "BOO"      Prints _ and "BOO" to the current terminal
READ 1,1: X         Reads Record 1 from Disc File 1
SCAN "DATA":        Resets the SCAN pointer to location "DATA"
```

Although the colon is almost always used in the above manner, it is quite likely that the programmer may find other uses for it. Imagination and experimentation can produce interesting results!

The Comma

Commas are non-executable statements used only to delimit parameters within a list. During compilation, all commas are deleted. If there is an obvious separation between two constants, the comma is not needed, but should probably be used nonetheless for clarification. Examples:

```
PRINT XYZ, ABC        (two variables XYZ and ABC)

PRINT XYZ ABC         (one variable XYZABC)

PRINT 1,A,"HORSE"
1   DOG   HORSE

PRINT 1A"HORSE"
1   DOG   HORSE
```

Peripheral Devices

OPUS will handle almost any combination of perpheral devices through the System Generation Routine. Two types of devices can be implemented:

1. Serial Devices

   Serial devices are those peripherals that sequentially send and receive one character at a time. Included in this category are terminals, paper tape readers and punches, cassettes, etc. In the System Generation Routine, the user defines the input and output drivers for these devices, with the routine automatically assigning a device number (1,2,. . .) to each device. By using this device number, the user can generate output or receive input with various I/O commands. Programs can be saved or retrieved from these devices (normally paper tape or cassette). Although OPUS will not internally handle data files on these serial devices, the user can nonetheless dump and retrieve data in a user-defined format.

2. Random Devices

   Random devices are normally assumed to be disc drives, in particular, floppy discs. Programs may be saved and retrieved. Data files may be created, allowing the user to randomly access any record within the file. The system generation will assign a disc number (Ø,1,. . .) to each random device.

   Although discs are the normal random devices, it is possible to enter other devices, such as software-controlled cassette units. These devices must be separated into sectors and tracks.

Data Files

All data files in OPUS are random files. No sequential file may be treated as data (i.e. program files, etc.). Each data file is divided into two parts:

1. The Map

   When a file is opened (OPEN command), the map portion of the file is created. The map requires enough continuous sectors on the disc to hold all possible sector pointers to the data sectors. Each pointer consists of two bytes: the track and the sector. As data is written in the file, the map is updated to point to the location of the data. The relative position in the map corresponds to the record number of the data.

2. The Data Sectors

   The actual sectors which will contain data are not allocated until data is written in the file. When data is to be written, OPUS will find a free sector on the diskette, write the record, and update the map to point to this location. Depending upon the number of bytes per sector, each sector may contain a varying number of logical records.

With the OPEN command, the user must specify the number of logical records that can be held in one sector and the maximum number of logical records the file may hold. A logical record may consist of a varying number of strings and numbers that define a specific piece of information. Records within the file are always referred to by the logical record number (1,2,. . .). Any record within the file may be referenced at any time with either a read or write command.

See Section VIII. for a more detailed look at data files.

Disc Tags

While each drive unit in the system is assigned a disc number, every data diskette may be assigned a disc tag. This tag is any string of ASCII characters, including alphabetical characters, numbers, control characters, etc., not to exceed a length of 7 characters. This tag is given to the diskette during the format procedure in the System Generation Routine or with the TAG command (OPUS/-TWO and OPUS/THREE only).

The tag allows the programmer to specify a diskette without being concerned with the particular drive. All disc commands that reference some file or program on a diskette may optionally use this disc tag, as a parameter, to specify upon which diskette the file is to be located. OPUS keeps a table in memory of all discs that have been enabled, along with the disc tag. This table is referenced each time a tag is declared. The following example loads the source from the diskette with tag "XYZ":

```
GET "PROG","XYZ";
```

The next one assigns a file from the diskette with tag "TEST":

```
ASSIGN "FILE",1,"TEST"
```

Disc Swap Routine

If a disc tag is declared for a diskette that either is not currently in a drive or has not been enabled (and thus not in the disc table in memory), OPUS will enter the Disc Swap Routine:

(TAG) DISC MUST BE LOADED; ENTER DRIVE NUMBER:

SWAP DISC & HIT RETURN

After printing the first prompt, OPUS will wait until the user responds with a drive number designating the drive into which the diskette with the given tag will be loaded. The user must enter the number (Ø,1,. . .) and hit carriage return. OPUS will then print the second prompt and pause. At this time, and only at this time, the user may remove the old diskette, if any, from the designated drive, and insert the new diskette with the correct disc tag. The user must then hit carriage return to inform OPUS that this has been accomplished. If this new diskette is the correct one, the program or operation will continue on; otherwise, the routine will be repeated.

This routine allows diskettes to be swapped in and out throughout the operation of a program. Note that any files assigned on a diskette which is removed will be automatically closed.

Although this routine appears simple enough, it is extremely important that it is followed to the letter. The old diskette must be removed and the new one inserted ONLY when OPUS is pausing after printing SWAP DISC & HIT RETURN. If the diskettes are changed after the first prompt, the sector maps of the diskettes will be written incorrectly, reflecting wrong data. At this point, the data must be transferred from these data diskettes to new ones or else the data will become increasingly unreliable (see Section XV.).

Enabling Discs

Discs are enabled by using either the DISC command or specifying a drive number in conjunction with a command that looks at the disc for some operation (GET, SAVE, ASSIGN, etc.). When enabling a disc, OPUS reads the sector-free table and the disc tag into memory. The sector-free table is essentially a map of all locations on the diskette that have been filled with data and those that are empty or available for data. Each diskette will have a different table according to what data has been written on it. This enabling will only occur once, after which all references to that diskette assume this table is correct.

As easily can be seen, many problems will arise if diskettes are casually swapped in and out of a drive without following the correct procedure. If the sector-free table is written back on the wrong diskette, data may be written over other data in the future. Again, we urge the programmer to refer to Section XV. for a description of preventive procedures.

Once a diskette has been enabled, this procedure must be followed to remove it and/or insert a new diskette:

1. Make sure all data files are closed (use the CLOSE command) if they have been previously ASSIGNed.

2. OPUS/ONE: Remove the current diskette

   OPUS/TWO: Use the SWAP command to remove the current diskette

3. Insert the new diskette

4. Enable the new diskette with the DISC command

To disable a disc, the user need only make sure all files are closed (CLOSE command).

Program and File Names and Types

Every program that is to be saved or loaded from either a sequential or random device and every file that is to be opened on a random device must be given a file name and a file type.

The file name is any string of ASCII characters not to exceed a length of seven. The name may include numbers, characters, or other keyboard characters. The backslash (\) must not be used within a name. The name may be specified simply as a string constant or as a calculated expression.

Each program or file is also given a file type designation -- one character describing the type of file:

S	OPUS source program
O	OPUS object program
F	OPUS data file
D	OPUS dimensioned data file (OPUS/TWO only)

Other software developed by A.S.I. will use other file types than these. Refer to Appendix C. for a more complete listing.

All statements requiring a file name as a parameter assume one of the above types as a default. However, it is possible to designate a different type by entering a backslash (\) and the type character after the file name.

```
file name \ type
```

The backslash and type must be included within the same string constant. OPUS will save or retrieve this file with the abnormal type designation. Examples:

```
GET "TEST" or GET "TEST\S"    Loads source program
ASSIGN "DATA",1               "F" file
```

```
ASSIGN "DATA\D",2          "D" file
DUMP "TEST\S"              SAVEs source program
SAVE "TEST"                SAVEs source program
KILL "PROG\0"              KILLs object program
```

Errors

There are essentially three types of errors that can occur in OPUS. All of them will occur during execution only. Because of the syntax flexibility, it is not possible to check for syntax errors during code entry and compilation.

a. Statement Errors

Many statements require their parameters to be within certain ranges or of certain format. If an expression value which is to be used by the statement does not follow the format rules for that statement an error will occur:

(STATEMENT) ERROR

Program termination will follow immediately.

No location is given for the error because no line numbers exist in the object code. This places a burden on the programmer, but by following some of the debugging hints below, the chore is not that difficult.

The manual does not list the possible errors for each statement. The programmer should assume that if a statement error occurs, the values (or lack of values) received by the statement do not satisfy the format requirements of the statement.

In OPUS/TWO and OPUS/THREE, statement errors may be supressed by the ERR command.

b. Buffer Overflow Errors

OPUS utilizes several memory buffers for various functions. If any of these buffers are overflowed, an error will occur and program termination will result. All variable values will be lost, although the program remains intact. These errors may not be suppressed by any means.

1) Operand Stack

Described in depth above, this buffer holds all operands to be used by following statements during execution. Only the memory address of the value and the type identifier (number, string, matrix) are actually stored in this buffer. Each operand therefore takes 3 bytes of memory. Possible error:

OPT OVERFLOW!

Meaning:

There is not enough room for all operands in the stack. The size of this buffer may be increased by using the SET command with parameter 1.

2) Constant Table

This buffer holds all temporary values of calculated expressions. For example, arithmetic operations will return a value that will go into this constant table. The value will be purged as soon as it is used by another statement. The size of this table is dependent upon the number of bytes required by these calculated values. Possible error:

CON OVERFLOW!

Meaning:

There is not enough room for the calculated values. The size of this buffer may be changed by using the SET command with parameter 2.

3) Variable Name Table

This buffer holds all the names of variables used within a program. The number of bytes required by each variable is the number of characters in the variable name plus two. The two extra bytes hold the memory address of the value of the variable in the variable value table. Possible error:

VNT OVERFLOW!

Meaning:

There is not enough room for all variables used. The size of this buffer may be changed with the SET command using parameter 3.

4) Variable Value Table

This buffer holds all the values of variables used within a program. The values are strings or numbers and the size of the table is determined by the size of these constants. As these constants will be constantly changing throughout the program, so will the size of this buffer. This buffer is located after the program area and will include all memory up to the function stack. Possible error:

VVT OVERFLOW!

The only way to increase the size of this buffer is by adding more memory or shortening the length of the program. If more memory actually exists in the system, the SET command may be used with parameter 7. Otherwise,

additional physical memory must be added.

5) Function Stack

This stack has a multitude of uses. It holds various values used by block statements, such as the return address for the RETurn statement. Compilation will use this stack temporarily. Each item in the stack requires three bytes, one to identify the use, and the other two to specify a memory location. Possible error:

STACK OVERFLOW!

There are two possible reasons for this error:

1. The stack has been pushed down to the point that it overflows into the variable value table. The only solution is to increase memory or shorten the program.

2. Some block terminator has been executed without executing a block initiator, such as executing a NEXT statement without a prior LOOP. The program must be corrected. The reason that an error occurs in this instance is that OPUS, upon reaching a block terminator, will scan the stack for the matching initiator. If it is not there, the stack is overflowed at the top.

6) Program Area

This buffer is reserved for user programs. Its length is variable, according to the amount of memory in the system. Possible error:

MEMORY OVERFLOW!

This will occur if a program is loaded or entered that overlaps memory boundaries. Memory must be increased.

c. Disc Errors

There are four types of disc errors:

1) File Errors

If an attempt is made to access a data record that does not exist in a disc file, an error will occur. If the record accessed is a valid record number but contains no data, and a READ or PURGE is attempted, an End-Of-Record error and program termination will occur:

EOR ERROR

If the record accessed does not fall within the maximum record boundary of the file, an End-Of-File error and program termination will occur:

EOF ERROR

Both these errors may be suppressed using the EFILE statement.

In OPUS/TWO and OPUS/THREE, if errors are trapped with the ERR statement, these two errors will also be supressed. The numerical value of each (equivalent to the statement number) is as follows:

EOF	126
EOR	127

These numbers are accessible by the "?" statement.

2) Overflow Errors

If a diskette reaches the point where no more data may be saved on it, this error will occur:

DISC OVERFLOW!

The program or file record will not have been successfully written on the disc. Either other data must be purged or a different diskette used. This message will also occur if the programmer attempts to open a file requiring more sectors to store the file map than are available on the diskette.

If the directory on a diskette becomes full, that is, there is no more room to store program and file names, this error will occur:

DIRECTORY OVERFLOW!

Again, the program or file will not be successfully created. Normally, at least 16Ø programs and files may be saved on one diskette, although this is dependent upon the type of disc drive implemented.

3) Disc Failures

Should a disc drive fail or a diskette become unreadable due to alignment, etc., the following errors may occur:

(D)/(T)/(S) DISC RETRY!

D:	Disc Number
T:	Track
S:	Sector

These numbers are in decimal in OPUS/ONE and OPUS/TWO, and in Octal or Hex in OPUS/THREE. This error may be printed if a read or write to the disc was unsuccessful the first time but did eventually succeed. Execution will continue normally. The user should be warned that something is not quite right with the drive.

The following error will occur if a read or write to the disc was completely unsuccessful.

(D)/(T)/(S) DISC FAILURE!

Program termination will result. If the track and sector numbers are valid, something is wrong with either the diskette or the drive. If the track and sectors are not valid, it means that the data on the diskette has been altered incorrectly (see Section XV.). It is recommended that the diskette be used as a read-only diskette and data transferred to a new diskette.

4) Miscellaneous Disc Messages

-NO DISC DECLARED-

A disc command was entered without any disc being specified. Either use a parameter to specify the disc or the DISC command.

Debugging Hints

Trying to find the bugs in a program is the programmer's number one headache. The following tips may prove to be of help.

1. In order to trace the flow of a program, it is often helpful to insert PRINT statements liberally throughout. The values of various variables can be printed to give some idea of what is happening. Also it can be determined if indeed a program is reaching certain points.

2. If a program bombs on a statement error, the programmer can print out, in command mode, various values of variables in an attempt to determine the source of the error.

3. If very strange things are happening in the order of execution, first check for matching parentheses and brackets. This can always be a source of problems.

4. Check for any miscellaneous values that may have accidentally been left on the operand stack and inadvertently used by the wrong statement.

5. Check for OPUS statement words that may have been used in a variable name. OPUS will attempt to execute the statement with a different variable than intended.

6. In OPUS/THREE only, there is a TRACE function which essentially single steps through a program, giving each instruction with the contents of the operand stack and function stack. Free memory can also be determined at any point.

Section XV. describes several common programming problems with possible solutions.

C. USING OPUS

Assuming that you have properly initialized and brought up your system, you are now ready to begin programming. This section is intended to help you get started.

First, it would be appropriate to address a general point about OPUS. Computers are extremely flexible machines; indeed, it is their beauty. However, there are certain limitations, or boundaries, within which the user must operate. The act of designing a high-level language defines the relative values to be placed on several variables within the boundaries set by the hardware. One of these variables is the flexibility offered to the programmer.

OPUS has been designed to maximize flexibility. This maximization, in and of itself, places a heavier burden on the programmer. That is, achieving high flexibility requires extra care. Clearly, the most flexible language is machine language, and the least flexible a language which imposes severe constraints through limited vocabulary or strict syntax requirements.

The point of this: OPUS was set up to try to combine maximum flexibility with ease of use for the programmer. At times it can be frustrating, for you may find yourself convinced that there is a serious 'bug' in the language, when in actuality, there is a flaw in your program. We would suggest, therefore, that you make every effort to assume a logical program flaw before contacting A.S.I. with your problem. We are more than happy to be of assistance, but we are also trying to develop and provide more software for you, which is a time consuming business.

To the terminals!

Again, assuming that the system has been initialized, that you have typed in answers to DAY?, MONTH? and YEAR?, you are ready to proceed. First, try something in command mode. This will be a one-time statement which is automatically compiled and executed. Try typing the following, then hit RETURN:

```
PRINT "THIS IS A TEST";
```

The machine should respond:

```
THIS IS A TEST

FINE
```

"FINE" simply means that your one-line program was accepted, has been compiled and executed, and that the system is now ready for more input. We can do the same thing in a program. Type:

```
10 PRINT "THIS IS A TEST";
```

Notice that the statement was preceded by a line number. OPUS interprets this to mean that the statement is a part (or all) of a program, and will not execute the line until so instructed. Now type:

```
COM
```

This is the COMpile command which converts your source code into object code, a form which is now ready to be executed. The machine should respond with FINE. Now type RUN. OPUS now executes the program and returns to command mode:

```
THIS IS A TEST

FINE
```

You have now written a program. Since the source code is still in main memory, along with the object code, it may be LISTed and edited. Note that the compiled version, or object code, may be RUN again and again, until a change is made, when the source program must be reCOMpiled before it may be RUN.

Now type LIST. The computer will print:

```
10 PRINT "THIS IS A TEST";
```

At this point the line may be edited or deleted. OPUS is a line-oriented language; that is, in order to change anything in the program, the entire line must be re-entered.

Now let's add to the current program. Type in another line like this:

```
20 INPUT "WHAT IS THE NUMBER? ", X;
```

When compiled, this will cause the computer to send WHAT IS THE NUMBER? and wait for the user to type a response. Now another line:

```
30 PRINT "THE SQUARE ROOT OF ", X, "IS: ", SQR(X);
```

Go ahead and COMpile the program, and RUN the object program like this:

```
COM

FINE

RUN
```

The computer responds:

```
THIS IS A TEST
WHAT IS THE NUMBER?
```

Type in any number, say 4, and push the return key. The computer prints:

```
THE SQUARE ROOT OF   4    IS:    2

FINE
```

Now suppose we wish to repeat this sequence several times. LIST the program, and add these lines:

```
5 "START";
40 GOTO "START";
```

Be sure that the "START" in line 5 is identical to the "START" in line 40. It's easy to slip on the keyboard and type a non-printing character, or add a space somewhere. If the two labels are not identical, an error will occur. Now COMpile and RUN:

```
THIS IS A TEST
WHAT IS THE NUMBER? 4
THE SQUARE ROOT OF    4   IS:    2
THIS IS A TEST
WHAT IS THE NUMBER?
```

Now you may want to stop the program. Since we didn't make any special provision for this, hold down the CONTROL key and type C, then push the RETURN key. The computer responds:

```
FINE
```

We can add a line now to take care of this problem:

```
25 IF NONE [END;];
```

This tells the computer that, if the user responds to an INPUT request with a CARRIAGE RETURN only, it should execute an orderly HALT. Try it. (Remember to COMpile before typing RUN).

Listing the program now shows that the line numbers are not as evenly spaced as would be nice. To remedy this, type REN, for RENumber. LIST the program again; it should be evenly numbered in increments of 10, starting with line 10.

As you can see with this simple example, writing a program can be thought of as simply building a structure with blocks of code. OPUS has been designed to facilitate this approach. You will notice that line 40 utilizes this block structure approach; instead of:

```
IF NONE [END ; ] ;
```

we could have said:

```
IF NONE [PRINT "FINISHED"; END;];
```

or any other group, or block, of code. Since compiling removes line numbers, one may even divide a block into several lines. For example:

```
10 INPUT A;
20 IF NONE
30 [PRINT "NONE";
40 END;];
```

This provides considerable flexibility for the programmer, and can make 'debugging' the program much simpler. Remember to match brackets, however. There must always be a right bracket for every left bracket in the program.

Probably the best way to learn a language is to try it. If your program runs correctly, you have written a working program; remember, though, that the best program is the one which does the most work with the least amount of code. Don't try to get too fancy to start with, but keep in mind that OPUS can be written very efficiently. Sample programs are shown in the manual. Studying them should help give you an idea of the kind of programming which may be accomplished with OPUS.

To continue on with OPUS, be sure to take maximum advantage of the rest of this manual. The advanced programmer will want to pay particular attention to the Statement Table in Section XVII.D.

D. MANUAL FORMAT

On the whole, each command in the manual is shown in the following format:

Command Name

----------Short Explanation----------
----------.

Format: command $expr_1$ < $expr_2$, $expr_3$ >

----------Complete Explanation----------

----------. Example:

```
10    ----------
20    ----------   Source Program
30    ----------
                .
                .
                .
      Executed Object Code
```

Two points should be noted here:

1. Carets (< >) placed in the Format section are used to denote optional parts of the command. For instance, the command LIST may be used by itself and need not include the five parameters available.

2. In the examples, we have deleted the commands COMpile and RUN and the system responses of FINE. These are implied with the three elisions between the source program and the executed code. Thus, the program format should be thought of as:

```
10    ----------
20    ----------
30    ----------

COM

FINE

RUN
Executed Object Code

FINE
```

III. COMMANDS

A. FUNDAMENTALS

Commands are OPUS statements that are primarily used in command mode to initiate some process or to edit source programs. Most of these commands will optionally use a list of parameters for execution.

Although most commands may be used within a program, there are two exceptions:

COMpile
RUN

If used within a program, the COM command will still compile source code to object code, but it will automatically return to command mode.

If used within a program, the RUN command will start execution of the object program at the beginning. However, if the RUN command is repeatedly executed, the machine code stack will quickly overrun, and it is possible to wipe out the operating system.

B. DEFINITIONS

COMpile

The COMpile command will cause the source program currently in memory to be put into object code format in preparation for execution.

Format: COM

The source program itself will not be affected by this operation. This command must be given prior to RUNning a source program because it is the object program that is actually executed. All operatins that are directly executed in the command mode will be automatically COMpiled before execution.

In OPUS/TWO and OPUS/THREE, ASCII files previously created by a text editor may be loaded as programs, COMpiled once to produce source code, and COMpiled a second time to produce boject code.

Format: COM < expr >

The expression must have the value "A" to COMpile ASCII to source or "S" to COMpile source to object. If not specified, source to object is assumed. The ASCII to source will automatically assign line numbers to each line of code. Thus, the ASCII file should not contain line numbers.

DELete

This command may be used to delete one or more lines from a source program.

Format: DEL $line_1$<,$line_2$>

Line 1 is the first line to be deleted and line 2 is the ending line number (exclusive -- all lines up to but NOT including this line) of the section to be deleted. If only line 1 is entered, OPUS will delete all lines from this number through to the end of the program. If only one line is to be deleted, it is easier to simply type the line number and hit the RETURN key.

SET

This command may be used to set buffer and parameter sizes:

Format: SET $expr_1$, $expr_2$

$Expr_1$	Description	$Expr_2$
1	Change Operand Table size	# bytes required
2	Change Constant Table size	# bytes required
3	Change Variable Name Table size	# bytes required
4	Change Input Buffer size	# char. per line
5	Change number of files to be assigned simultaneously	# files
6	Change digit accuracy	# digits
7	Change memory size	# bytes

The SET command may be used within a program; however, all variables previously entered will be reset.

LIST

This command will list the source program currently in memory.

Format: LIST<$expr_1$:><$expr_2$, $expr_3$, $expr_4$, $expr_5$>

Expression 1 is the output device number to which the listing is to go. Default declares the current device as the correct one. Expression 2 is the starting line number of the section of program to be listed. Default causes the list to start at the beginning of the program. Expression 3 is the ending line number (exclusive -- up to, but not including, this line number) of the section of program to be listed. Default causes the list to continue to the end of the program. Expression 4 is the number of lines to be listed per page. Sixty lines is the normal number of lines for an eleven-inch page. Default causes the program to be listed without paging.

Expression 5 is the number of line feeds to be generated between every page of the listing. The default is 6, which will generate 6 line feeds. If expression 4 is not specified, expression 5 will be ignored. It is not necessary to enter all five parameters if only the first few are required. OPUS will assume default values for all those that are not entered. Zeros may be entered to specify the dafault values.

NEW

The NEW command clears the program buffer.

Format: NEW <expr>

The expression may be one of the following: "S", which clears the source program area only, or "O", which clears the object program area only. If no expression is entered, all program areas are cleared.

In OPUS/TWO and OPUS/THREE, if an ASCII file has been loaded (previously created by a text editor), the NEW parameter may have one more value: "A", which clears the ASCII program area only.

RUN

The RUN command is used to start program execution.

Format: RUN

It will always start program execution at the beginning of the program. Only object programs may be RUN, and if no object program is in memory, an error will occur. The object program may be created from source code with the COMpile command, or it may be loaded from a peripheral device.

It is possible to start program execution within the middle of a program by executing a GOTO statement in command mode to a specified label within the program.

RENumber

The RENumber command may be used to renumber line numbers in a source program.

Format: REN <$expr_1$, $expr_2$, $expr_3$, $expr_4$>

Expression 1 is the new line number with which the section of program is to begin. Default is line number 10. Expression 2 is the interval between line numbers that will occur in the renumbered section. Default is 10. Expression 3 is the current line number of the start of the program code to be renumbered. Default is the start of the program. Expression 4 is the current line number of the end of the program code to be renumbered. This line number is excluded from renumbering. Default is the end of the program. The value of all expressions must be integers greater than or equal to zero. If no parameter is entered, or if the parameter is zero, the default value will be assumed.

```
1    A= 9;
2    B= 2;
3    C= 4;
4    PRINT A, B, C;

          .
          .
          .
REN

          .
          .
          .
10   A= 9;
20   B= 2;
30   C= 4;
40   PRINT A, B, C;
```

BYE

This command will update all discs in use and terminate OPUS operation.

Format: BYE

The computer types "---OPUS TERMINATED---" and will no longer respond. When the user is finished with OPUS, he should type BYE and halt the computer or turn it off.

In OPUS/THREE, running under the TEMPOS Operating System, the BYE command will cause the job to be terminated, not the master operating system.

IV. ASSIGNMENT

A. FUNDAMENTALS

Assignment is the action of giving a variable a specific value. The value may be either a string or number constant. An operation involving the assignment statement, "=", places the variable which receives the value on the left-hand side of the "=", and the value itself on the right-hand side. The value may be calculated from any operation(s). However, only one operand must be returned to the operand table to be assigned to the variable.

The variable which receives the value may be one of three types:

Simple variable
Matrix variable element
Substring

A simple variable is a character name that has a value, number or string. The name must consist of only upper-case alphabetical characters, with no limit to the length. A variable is always given an initial value of zero, prior to program assignments.

The format of the matrix variable elements and substrings are given in Sections XII. and VI., respectively.

It should be stressed that any variable may be assigned either a number or string value. OPUS internally keeps track of the type of constant.

After completion of the assignment statement, the value that was assigned will be returned to the operand table for optional use by another statement.

Assignment: =

The equal sign is the symbol for assignment.

Format: variable = expression

The value of the expression on the right of the equal sign is assigned to the variable on the left. Remember that this operation does not mean "equals", but rather "is given the value of". The variable must be a simple variable, a matrix variable element, or a variable substring.

Simple Variable

```
X= X - 3;
```

X is given the value of the expression X - 3 .

Matrix Variable Element

```
M(2,5)= "HORSE";
```

The value of "HORSE" is assigned to the matrix element (see Matrices).

Variable Substring

Assume ST has the value "CAT". By executing:

```
ST$(3,3)= "R";
```

ST will now have the value "CAR" (See Substrings).

In the following example, "HORSE" is assigned to "A", returned to the operand table, and used by the PRINT statement.

```
PRINT (A="HORSE");
```

It is important to note that the value of the right-hand expression is returned to the operand table for optional use by another statement. If this is not needed, be sure to execute a semicolon immediately after the assignment operation to clear the operand stack.

If one needs to assign the same value to several variables, instead of

```
A=Ø; B=Ø; C=Ø;
```

it is possible to execute

```
A=(B=(C=Ø));
```

The "=" sign may not be used as a relational operation for comparing the first value against the second. The programmer must use the operation IS for this purpose.

V. ARITHMETIC OPERATIONS

A. FUNDAMENTALS

The following arithmetic statements are binary operators requiring two numerical values for execution:

Operator	Priority	Description
↑	11	Exponentiation
/	10	Division
*	10	Multiplication
-	9	Subtraction
+	9	Addition

The statement will remove these two values from the operand stack, perform the operation, and return the resulting value to the operand stack for use by another operation.

The following arithmetic statement is a unary operator, requiring one numerical value from the operand stack and returning one numerical value to the operand stack:

Operator	Priority	Description
-	12	Negate the operand

As can be seen, the key word for both negation and subtraction, "-", is the same. During compilation, OPUS will determine the way it is to be used by determining if one or two operands are available for the operator.

If an operand is received in string format, OPUS will automatically convert the value to numerical format before proceeding with the operation. The programmer need not be concerned with this format.

The priority numbers given above with each operator determine the order of execution when more than one operator is included in an expression. The numbers with higher values indicate that these operators will have higher priority over the operators with lower priority numbers. For example:

```
1+2*3-16/4↑2
```

will be executed as if parentheses were positioned in this manner:

```
1+(2*3)-(16/(4↑2))
```

The value would be:

```
1+6-16/16
7-1
6
```

Parentheses may be used at any time in an arithmetic expression to clarify the order of execution. By using parentheses differently than in the above expression, the resulting value could be completely different:

```
(1+2)*3-(16/4)↑2
```

would produce:

```
3*3-4↑2
9-16
-7
```

Note that if two operators have the same priority number, the order of execution is determined by the order in which they appear in the expression. The one on the left will be executed first, followed by the one on the right.

There is no limit to the size of a single arithmetic expression other than the size of the operand stack and constant table (which can be enlarged).

As many parentheses as desired may be used within an expression, nested to any depth. However, be sure that the number of left parentheses match the number of right parentheses. OPUS will not give an error if they do not match, but the order of execution will be altered significantly.

B. DEFINITIONS

Addition: +

The "+" sign is the arithmetic binary operator for addition.

Format: $expr_1 + expr_2$

The value returned will be the sum of the two expressions.

```
PRINT 165+99
264
```

Subtraction and Negation: -

The "-" symbol may be used as either the unary operation of negation, or the binary operation of subtraction, depending upon whether one or two operands are present.

Format: $-expr_1$

This format returns the negated value of the expression.

Format: $expr_1 - expr_2$

This format returns the numerical difference between expression 1 and expression 2.

```
PRINT -6,6-2,-(A*3)
-6   4   -30
```

Multiplication: *

The asterisk is the arithmetic binary operator for multiplication.

Format: $expr_1$ * $expr_2$

The value returned by the operation will be the product of the two expressions.

```
PRINT 135*22
2970
```

Division: /

The slash is the binary operator for division.

Format: $expr_1$ / $expr_2$

The value returned is the quotient of expression 1 (the dividend) and expression 2 (the divisor). It should be noted that if the quotient is a continuing fraction, it will be carried out to the maximum number of digits as allowed in the precision declaration. The last digit will be rounded to the nearest unit.

```
X=1/3;PRINT X
.33333333
```

Exponentiation: ↑

The up-arrow is the binary operator designating exponentiation.

Format: $expr_1 \uparrow expr_2$

The value of expression 1 is raised to the power of the value of expression 2. The result is returned to the operand table. In OPUS the expression X^2 is written as X↑2. Note: Precision will be carried out to 6 digits maximum.

VI. STRING OPERATIONS

A. FUNDAMENTALS

A string is a sequence of ASCII characters with a length from Ø through 127 characters. Any variable or matrix element may be given a string value. It is not necessary, nor is it possible, to dimension variables or matrices to hold string constants. OPUS does not care whether a string or numerical constant is assigned to a variable.

If it is necessary to include a reserved character within a string, such as Control **X** or a quote, the ASC function should be used.

If an operation requiring a string value is encountered, OPUS will automatically convert any numerical values into string constants prior to the execution.

A null string is a string with no characters. Its string constant value is designated as: "". Null strings may be used in any operation requiring string values. If a null string is converted to numerical form, its value will be Ø.

Portions of a string (substrings) may be easily referenced or assigned values. The particular subset must be specified by declaring the starting and ending character positions. Numbers will be converted to string format during substring operations (see the "$" statement). Examples of strings:

```
"THIS IS A STRING OF LENGTH 29"
"THIS CONCATENATION WITH A NULL STRING PRODUCES THE ORIGINAL STRING" & ""
"123.333333"
"!!!!#####&&&&&"
```

B. DEFINITIONS

Quotation Mark: "

The quotation mark (") is used to delimit a literal string of characters. The user must enter a quotation mark to determine the start of every string, but if the user does not type a quotation mark following the end of the string, OPUS will assume that the end of the line of input is the end of string. For example:

```
GET"TEST

FINE
```

is equivalent to:

```
GET"TEST"

FINE
```

Concatenation: &

The "&" symbol is the binary operator for string concatenation.

Format: $expr_1$ & $expr_2$

Concatenation is the process by which one string is added to the end of another string. The value returned will be the value of expression 1 immediately followed by the value of expression 2. This concatenated result will always be a string.

```
10   INPUT "FIRST WORD?", A, "  SECOND? ", B;
20   PRINT "CONCATENATED:", A & B;
       .
       .
       .
FIRST WORD? HORSE  SECOND? CART
CONCATENATED:   HORSECART
```

Substring: $

The "$" symbol must be used to specify a substring.

Format: variable $ (expr$_1$,<expr$_2$>)

The variable must be either a simple variable or matrix variable element. Expression 1 of the substring is its first character position within the string. The number must be an integer from 1 to the total number of characters in the string, where 1 is the first character in the string, 2 is the second character in the string, etc. Expression 2 of the substring is its last character position within the string. The number must be an integer from 1 to the total number of characters in the string, where 1 is the first character in the string, 2 is the second, etc. It is not necessary to enter the ending position; if no value is determined, it is assumed that the end of the string is the end of the substring. The ending number must be greater than or equal to the starting number of the substring.

Substrings may be used within a list or wherever an expression value is requested. They may be assigned a value with the "=" statement. In the following examples of substrings, assume X is "THIS IS A STRING".

```
X$ (1,4)    is "THIS"
X$ (6,6)    is "I"
X$ (11,16)  is "STRING"
X$ (11)     is "STRING"
```

In the next examples, X is .3456:

```
X$ (1,1)    is "."
X$ (4,5)    is "56"
X$ (1)      is ".3456"
```

To reference a substring of a matrix element, assume a two-dimensional matrix, M, where M(2,3) has the value "1A2B3C". Then:

```
M(2,3)$(2,4) is "A2B"
M(2,3)$(6) is "C"
```

If a substring is to be assigned a value, the number of characters affected in the string will be the lesser of the length of the specified substring and the assigned value.

Let X = "ABCDEFG"; the following assignments will make these changes to X:

```
X$(1,3) = "XYZ"          X:  "XYZDEFG"
X$(1,3) = "*"            X:  "*BCDEFG"
X$(5,5) = "BOO"          X:  "ABCDBFG"
X$(3,3) = ""             X:  "ABCDEFG"
```

VII. INPUT/OUTPUT OPERATIONS

A. FUNDAMENTALS

The statements included in this section all deal with either sending characters to a peripheral device (output) or receiving characters from a peripheral device (input).

All input/output statements optionally use device numbers to determine the source of input or output. Each serial device on the system, including terminals, cassettes, and paper tape units, are assigned a device number during system generation. The device numbers start with 1 and increment sequentially.

If a device number is not specified in the statement, the current terminal will always assume the default value.

The statements in this section include commands to send or receive string or number data, and commands that will save or load programs from a serial device. The latter commands will normally be used only with cassettes and paper tape units.

B. DEFINITIONS

INput

This command transfers input control from one device to another.

Format: IN expr

The expression must be the valid device number of the device that is to have input control. OPUS will now accept input only from this new device. Should input control be sent to a device that is not currently on the system, an error will result.

INPUT

The INPUT command requests data to be entered from some perpheral device, which is then assigned to a variable(s).

Format: INPUT <device #:> list

The device number must be specified if input is to be received from a peripheral other than the current input device. Any valid device number (as defined during system initialization) is allowed. The list is a list of variables, constants, or expression values. OPUS will scan the list from left to right and operate according to the type of parameter:

> Variable: OPUS will request input (assumed to be in ASCII characters) from the device. It will accept data, character by character, until a carriage return is received. It will then convert the line of data into valid string format and assign it to the variable. It is important to remember that all input is in string format, even if a number is entered.
>
> Constant/Expression Value: OPUS will print the value to the current output device and continue scanning the list. It should be thought of as an INPUT statement containing a PRINT statement.

No carriage return/line feed will be sent to the device until the entire list has been scanned and appropriately executed. Thus, it is possible to enter several variable values on one line. If a carriage return is received by OPUS and no data has been entered, the INPUT operation terminates (even though there may be more variables in the list) and program execution continues to the next operation. The value of the variable that was to be assigned the line of input will remain unchanged, i.e., it will have the same value after the INPUT operation as it did before the operation. The NONE function may be used to determine whether or not data was entered.

```
INPUT A,B; IF NONE  [PRINT "NONE"; END; ];
```

This example requests input for A; when a carriage return is received, it requests input for B. If a 2nd carriage return is entered, the program types 'NONE' and terminates.

```
10   INPUT "TYPE SOMETHING: ", A; PRINT A;
                        .
                        .
                        .
TYPE SOMETHING: TRTRTRTR
TRTRTRTR
```

PRINT

The PRINT statement is used to send data to a specified output device.

Format: PRINT<device #:> <list>

The device number is the number of the desired output device. If none is given, the output device is assumed to be the current device. The list is any combination of variables, constants, or expressions. OPUS will scan the list from left to right, printing the value of each (in ASCII characters). It will print three spaces after each value and, at the end of the list, a carriage return/line feed will be generated only. There is no way to format the list differently (fewer or more spaces between values, suppress carriage returns, etc.). The programmer should use the Print Formatted (PF) statement for this.

```
10   INPUT "TYPE SOMETHING: ", A;
20   PRINT A; PRINT "TEST"; PRINT 1+ 2;
                  .
                  .
TYPE SOMETHING: THTHTH
THTHTH
TEST
3
```

Note that a matrix variable may be part of the list.

```
10   DIM  A(5);
20   LOOP K, 1 to 5;
30   A(K) = K;
40   NEXT;
50   PRINT A;
       .
       .
       .
1   2   3   4   5
```

OUTput

The OUTput statement transfers output to a designated device.

Format: OUT expr

The expression must be a valid device number as declared in the system initialization. After this operation is executed, any output from the computer will go to this device. This does not affect the input device.

Print Formatted

The PF statement is used to control the format of data sent to output devices.

Format: PF <device #:>expr<, list>

The device number is the number of the desired output device. If none is specified, it is assumed that the device is to be the current output device. The list is one or more expression or variable values that are to be generated to the output device according to the format string. The expression must return a string value, which is the format string. The following symbols may be used in any combination to compose the format string:

Bn — n blanks (spaces) are to be printed. If n is not given, one space will be printed.

Ln — The corresponding value in the list of the statement will be left-justified in a field of n characters. If the length of the value is greater than n, the value will be truncated to fit into the field. If the length of the value is less than n, spaces will be generated to fill out the field.

Ln.m — The corresponding value in the list will be scanned for a decimal point. If one is found, all digits or characters to the left of the decimal point will be left-justified in the field of n characters. The decimal point will then be printed. All digits or characters to the right of the decimal point will be left-justified in the field of m characters, with trailing zeros filling out the field. This field will be printed after the decimal point. If no decimal point is found in the value, all digits or characters will go into the n field, followed by a decimal point and zeros in the m field. Spaces will fill out the n field if necessary.

Rn — The corresponding value in the list will be right-justified in a field of n characters. If the length of the value is less than the n field, spaces will be filled in to the left of the value. If the length of the value is greater than the n field, the value will be truncated on the left.

Rn.m — The corresponding value in the list will be scanned for a decimal point. If one is found, all digits or characters to the left of the decimal point will be right-justified in the field of n characters. The decimal point will then be printed. All digits or characters to the right of the decimal point will then be left-justified in the field of m characters, with trailing zeros filling out the field. This field will be printed out after the decimal point. If no decimal point is found in the value, all digits or characters will go into the n field, followed by a decimal point and zeros in the m field. Spaces will be filled in to the left of the value in the n field if necessary.

S — The corresponding value in the list will be printed "straight", i.e., the field will be the actual length of the value -- there will be no spaces on either side of the value.

/ — OPUS will immediately send a carriage return and line feed to the output device. It is important to remember that after the PF statement is completed, NO carriage returns or line feeds are automatically generated. Thus, the slash is normally inserted at the end of the format string to force a carriage return/line feed.

\- — A carriage return will be sent to the output device; there will be no line feed.

\+ — A line feed will be sent to the output device; there will be no carriage return.

n(...) — All format symbols enclosed in the parentheses will be executed n times.

, — Commas may be optionally inserted between format symbols for clarity. OPUS ignores them.

In all of the above, n and m must always be integers greater than Ø but less than 128.

In the format string, these symbols with their parameters may be combined in any manner. OPUS scans the format string from left to right while simultaneously scanning the list values from left to right. If a symbol in the format string is found that does not require a list value, it is executed immediately. If a list value is required, the next value in the list is formatted to the particular specifications and output as ASCII characters to the specified output device. There must be at least the same number of values in the list as there are symbols in the format string requiring values, or else an error will occur. If there are more values than are needed, they will be ignored.

```
10   A= "QQQ";
20   PF "L3,R30", A, A;
          .
          .
          .
QQQ                          QQQ
```

or

```
10   A= 1234.567;
20   PF "R6.2/", A;

          .
          .
          .
  1234.56
```

Note that a matrix variable may be part of the list to be output.

LINe

The LINe command will generate a certain number of line feeds to a specified output device.

Format: LIN <expr$_1$>: expr$_2$

Expression 1 is the device number to which the line feeds are to be sent. If not entered, the current output device is assumed. Expression 2 is the desired number of line feeds and must be an integer greater than zero. This command should not be included in a PRINT statement. It will generate the line feeds as a stand-alone operation.

SPAce

This function will cause a given number of spaces to be immediately generated on the output device.

Format: SPA<$expr_1$:> $expr_2$

Expression 1 is the device number on which the spaces are to be printed. If not specified, the current output device is assumed. Expression 2 is the number of spaces to be printed and must be a positive integer less than 256. OPUS will generate these spaces (blanks) to the output device immediately upon execution of this SPAce function.

```
10  INPUT "NUMBER? ", A;
20  SPA A; PRINT "*";
              .
              .
              .
NUMBER? 30
                                *
```

SAVE

The SAVE command must be used to store a source program on a peripheral device.

Format: SAVE <device #:> expr

The device number is the number of the device on which the program is to be stored (i.e., cassette, paper tape, etc.). If no device number is specified, it is assumed the device is the disc (see Section VIII). The expression must be the name of the program -- any ASCII string not exceeding 7 characters in length.

```
SAVE 4: "TEST1"
```

Compiled SAVE

This command may be used to save a compiled (object code) program on a peripheral device.

Format: CSAVE <device #:> expr

The device number designates the serial device upon which the program is to be saved, normally cassette or paper tape. If this number is entered, it must be followed by a colon. If no device is specified, it is assumed that the program is to be saved on disc (see Section VIII). The expression is the name of the program, not to exceed 7 characters.

If a program is saved in its compiled form, it may not be recompiled prior to execution. Use the LOAD command to retrieve the object program from the peripheral device.

```
CSAVE 3: "TEST2"
```

GET

This command must be used to retrieve a source program from a peripheral device.

Format: GET <device #:> expr

The device number is the number of a peripheral device, normally cassette or paper tape, from which the program is to be retrieved. If no device number is given, it is assumed the program is to be loaded from the disc (see Section VIII). The expression is the name of the program, previously given to the program with the SAVE command. The name may be up to seven ASCII characters in length. The GET command will act as an append if another program is in core, i.e., it will add the new program to the end of the current program. Therefore, it is best to execute a NEW command to clear the program buffer before executing this command.

```
GET 4:  "TEST1"
```

LOAD

This command must be used to retrieve object programs from a peripheral device.

Format: LOAD<device #:> expr

The device number determines from which device the object program is loaded. If no number is given, the program is LOADed from the disc (see Section VIII). The expression is the name of the program previously assigned when it was SAVEd. The length may be no more than seven characters. The LOAD command will act as an append if another object program is currently in memory, i.e., it will add the new program to the end of the old.

```
LOAD 4:  "TEST2"
```

VIII. DISC OPERATIONS

A. FUNDAMENTALS

Disc operations include all statements that in some manner will affect data stored on diskettes. The cassette/paper tape version of OPUS/ONE will not contain any of these commands.

Disc Layout

A diskette used by a floppy disc drive is divided concentrically into physical tracks, with each track divided into sectors. The number of sectors and tracks is dependent upon the type of drive being used. Each sector holds a certain number of bytes of data, most commonly, 128. OPUS will handle any disc drive with the number of tracks, sectors, and bytes less than 256.

The first track of the disc is reserved by OPUS for a directory of all programs and files. The directory (or library) keeps the file name, the location on the disc, and other pertinent parameters that have to do with the file type. Also included on the first track is the sector-free table which keeps track of all sectors that are being used for data and those that are available.

The rest of the disc is available for programs and files. Programs are stored in a sequential manner with each sector in the program pointing to the next sector. Data files are treated by a random-access method. The file consists of two portions, the map and the actual data sectors. The map is in a contiguous block on the disc and thus, by implementing a simple algorithm, it is possible to directly access any portion of this map. The map contains pointers to the sectors holding the data. These sectors may be located anywhere on the disc. Appendix C. gives a much more detailed explanation of the disc layout.

Disc Formatting

Before any data can be stored on a disc, the disc must be formatted to OPUS specifications. The formatting routine is in the System Generation Routine. The primary function of formatting is to put zeros in all the sectors on the first track, therefore declaring that there are no files and that all sectors are available for data. Optionally, the format routine will zero out the rest of the diskette, checking primarily for bad sectors.

Disc Tag

With the format routine, it is also possible to assign a tag or label to each diskette. The tag may be any sequence of ASCII characters not to exceed a length of 7. The tag may be used by the following commands:

SAVE
CSAVE
DUMP
GET
LOAD
KILL
ASSIGN
OPEN
LIB

The tag determines which disc is to be accessed by the command. By specifying the tag as a parameter, the diskette may reside in any drive, and if it is enabled, OPUS will find it.

Note that if a tag is specified and either the diskette is not in the system or has not been enabled, OPUS will enter the Disc Swap Routine to allow the user to insert the correct diskette.

Disc Numbers

The above commands may also specify a disc number as a parameter instead of a tag. These disc numbers are assigned during the System Generation Routine. The numbers will start with Ø and increment sequentially. Normally the numbers will directly correspond to the physical drive number. However, in OPUS/TWO and OPUS/THREE, it is possible to have different sets of drives in the system; in which case, the disc numbers will be different.

Disc Swap Routine

If a disc tag has been specified in a command and the diskette is not present, OPUS will enter the following routine:

(TAG) DISC MUST BE LOADED - ENTER DRIVE #:

> Enter the number of the disc where the diskette is to be inserted. DO NOT remove the old diskette at this time. All files will be closed on this old diskette at this time.

SWAP DISC & HIT RETURN

> Remove the old diskette (if any) now and insert the new one. Hit carriage return to designate that this has been accomplished.

If the new diskette does have the correct tag, the operation will continue. If not, the routine will be repeated. It is possible to terminate the routine by typing Control C.

By taking advantage of this Disc Swap Routine, the programmer can write programs that may implement several diskettes. Diskettes can be swapped in and out easily by the user of the programs.

NOTE: In OPUS/THREE, running under the TEMPOS Multi-User Operating System, if this disc swap routine is entered and another user is using the disc specified in the first prompt, OPUS will return with:

DISC IN USE BY JOB XX

The routine will be repeated until a drive is declared that is free.

Removing and Inserting Diskettes

Once a disc has been accessed by the DISC command or some other command specifying a disc number, the disc is considered to be enabled. It must not be removed and another diskette inserted without heeding the following precautions. Failure to follow these procedures may result in a lost cause disc!

To remove a diskette:

1. Use the CLOSE command to make sure all files are closed on the disc.

2. OPUS/ONE: Remove the diskette
   OPUS/TWO: Execute the SWAP command to remove the disc

To insert the diskette:

1. Insert the new diskette
2. Execute the DISC command to enable this disc

Never remove an old diskette and insert a new diskette without notifying OPUS in this manner (see Section XV. for the details on how to mess up a disc!).

B. PROGRAM STORAGE & RETRIEVAL

Both source and object programs can be saved and loaded on discs. The programs are stored as sequential files. The program must be given a name consisting of not more than 7 ASCII characters. The type designation may be optionally included with the name. The types for OPUS programs are:

S	Source program
O	Object program

To include the type in the name, use the following format:

```
"NAME\T"
```

where T is the type.

The following commands are used to save programs:

SAVE	Saves a source program
CSAVE	Saves an object program
or	
DUMP	

These commands are used to load programs:

GET	Loads a source program
LOAD	Loads an object program

Program files may not be accessed as data from a program. Only random-access files can hold accessable data.

Programs are appended in memory unless the NEW command is used before each GET and LOAD.

SAVE

The SAVE command must be used to store a source program on a peripheral device.

Format: SAVE <device #:> $expr_1$ <,$expr_2$ >

The device number is the number of the serial device on which the program is to be stored (i.e., cassette, paper tape, etc.). If no device number is specified, it is assumed the device is the disc. Expression 1 must be the name of the program -- any ASCII string not exceeding 7 characters in length. If the program is saved on disc, there may be no other program or file on the disc with the same name. If the user has made changes to a program that was previously saved on disc, and wishes to save the updated version with the same name, the program must first be KILLed and then SAVEd.

Expression 2 defines the disc drive, if the device specified is the disc. This may be one of the following:

a. drive number from Ø to n, where n + 1 is the number of drives on the system, or

b. disc tag (an ASCII character string identified with the diskette during the disc format procedure in the System Generation Routine).

If the diskette is not present, or the drive containing the diskette was not previously enabled, OPUS will enter the Disc Swap Routine. If expression 2 was not entered, OPUS will assume that the diskette is in the drive last declared by the DISC command.

```
SAVE "TEST1"
```

Compiled SAVE or DUMP

This command must be used to save a compiled (object code) program on some peripheral device.

Format: CSAVE <device #:> $expr_1$ <,$expr_2$>

or

DUMP <device #:> $expr_1$ <,$expr_2$>

The device number designates the peripheral device upon which the program is to be saved, normally cassette or paper tape. If this number is entered, it must be followed by a colon. If no device is specified, it is assumed that the program is to be saved on disc. Expression 1 in the format is the name of the program, not to exceed 7 characters in length. Expression 2 defines the disc drive, if the device specified is the disc. This may be one of the following:

a. drive number from Ø to n, where n + 1 is the number of drives on the system, or

b. disc tag (an ASCII character string identified with the diskette during the disc format procedure in the System Generation Routine).

If the diskette is not present, or the drive containing the diskette was not previously enabled, OPUS will enter the Disc Swap Routine. If expression 2 was not entered, OPUS will assume that the diskette is in the drive last declared by the DISC command.

Use the LOAD command to retrieve the object program from the peripheral device.

```
CSAVE "TEST2"

     or

DUMP "TEST2"
```

GET

This command must be used to retrieve a source program from a peripheral device.

Format: GET<device #:>expr$_1$<,expr$_2$>

The device number is the number of some peripheral device, normally cassette or paper tape, from which the program is to be retrieved. If no device number is given, it is assumed the program is to be loaded from the disc. Expression one is the name of the program, previously given to the program with the SAVE command. The name may be up to seven ASCII characters in length.

Expression 2 defines the disc drive, if the device specified is the disc. This may be one of the following:

a. drive number from Ø to n, where n + 1 is the number of drives on the system, or

b. disc tag (an ASCII character string identified with the diskette during the disc format procedure in the System Generation Routine).

If the diskette is not present or the drive containing the diskette was not previously enabled, OPUS will enter the Disc Swap Routine. If expression 2 was not entered, OPUS will assume the diskette is in the drive last declared by the DISC command.

The GET command will act as an append if another program is in core, i.e., it will add the new program to the end of the current program. Therefore, it is best to execute a NEW command to clear the program buffer before executing this command. The GET command should not be used within a program; it will cause program termination and a return to command mode.

```
GET "TEST1",Ø
```

LOAD

This command must be used to retrieve object programs from a peripheral device.

Format: LOAD <device #:> $expr_1$ <,$expr_2$>

The device number determines from which device the object program is loaded. If no number is given, the program is LOADed from the disc. Expression 1 is the name of the program previously assigned when it was CSAVEd. The length may be no more than seven characters. The LOAD command will act as an append if another object program is currently in memory, i.e., it will add the new program to the end of the old.

Expression 2 defines the disc drive, if the device specified is the disc. This may be one of the following:

a. drive number from Ø to n, where n + 1 is the number of drives on the system, or

b. disc tag (an ASCII character string identified with the diskette during the disc format procedure in the System Generation Routine).

If the diskette is not present, or the drive containing the diskette was not previously enabled, the Disc Swap Routine will begin.

LOAD "TEST2"

C. DATA STORAGE & RETRIEVAL

Data Files

All data files in OPUS must be opened as random-access files. Each file is divided into logical records which may be accessed (read from or written into) in any order. Any record may be directly accessed without having to reference others. OPUS disc files have the following attributes:

1. All references made to the file are by the file number.
2. All references made to data within the file are by logical record number. This number starts with 1 and increments sequentially.
3. The length of the logical record (number of bytes that it contains) is user-determined when the file is created; the maximum is the number of bytes in a sector. All logical records within the file will be the same length.
4. Data is written in a logical record either as a string or number item. As many items as will fit may be written in one record.
5. Logical record boundaries may not be crossed by either a READ or WRITE command.
6. When a file is created, no disc sectors are allocated for the logical records. Only when a record is written does OPUS allocate an empty sector.
7. Sequential files must be set up as random-access files -- within the program, logical records must be sequentially addressed.
8. Programs may not be saved in a data file. They are treated in a completely different manner by OPUS and may not be accessed as data.
9. To change one item in a logical record, it is necessary to read the entire contents of the record, change the item, and write all items back again.
10. An end-of-record marker is written after the last item in a record.

File Name

Every file created on the disc must be given a name for future reference. The name may be any string of characters (from ASCII decimal code 1 to 254) not exceeding 7 characters in length. These data files have a type "F" designation, which must be used in conjunction with the KILL command.

File Number

The file number is a number temporarily assigned to a file in the ASSIGN statement. This file number, instead of a file name, will be used to reference (READ or WRITE) the file. During system generation, the user declares how many files may be assigned simultaneously in a program. This will determine the maximum valid file number. The file number is an integer greater than zero, but less than or equal to this maximum. It makes absolutely no difference which files are assigned what file numbers. In fact, files may be assigned to different file numbers during execution of the program.

File Access

Data files may be created on the disc to hold numbers, strings or matrices. To facilitate understanding, the following example shows how a simple data base may be created and accessed.

The user wishes to set up a data base from which to generate mailing labels for letters. It is decided that a disc file is the logical medium for such a data base. How should this be set up and implemented?

1. Define the logical record. Every disc file is broken down into logical records, each of which corresponds to a specific item in the data base. In this instance, the item would be a person to whom a letter will be sent.

2. Define the contents of the logical record, that is, define what information should be kept on each person, for example, a name and address. The address should be separated into street address and city/state because the two parts must be printed on separate lines.

3. Define the length of the logical record. How many bytes are required to store a name and address? Assume that a name will never be longer than 16 characters, the street address no more than 25 characters, and the city/state no more than 20. Because these items must be stored as strings, the name will require 17 bytes, the street address 26 bytes, and the city/state 21 bytes. Thus a total of 64 bytes will be required in the logical record.

4. Determine the maximum number of logical records. How many people must there be in the mailing list? Let us assume there will be 1500.

5. Open the file to the required specifications.

```
DISC Ø
FINE
OPEN "MAIL",2,1500
FINE
```

The DISC command enables drive Ø; if another drive holds a tagged diskette, this tag may be used in lieu of the drive number. "MAIL" is the file name. The second parameter, 2, gives the number of logical records in each sector. A sector contains 128 usable bytes; since the logical record requires 64 bytes, there can be 2 logical records in each sector (the integer of the number of bytes per sector, divided by the number of bytes per logical record). The third parameter, 1500, is the maximum number of logical records determined in step 3. The file has now been created, and data may be written in it.

6. Write a program to enter the data.

```
10   ASSIGN "MAIL",1; POS=1;
20   "START";
```

6. Write a program to enter the data (cont.).

```
30   INPUT "NAME? ", NA; IF NONE [ CLOSE; END ] ;
40   INPUT "STREET ADDRESS? ", ST;
50   INPUT "CITY, STATE? ", CS;
60   WRITE 1, POS: NA,ST,CS;
70   POS=POS +1;
80   GOTO "START";
```

The file "MAIL" is first assigned and given a file position number (ASSIGN "MAIL", 1). All READs and WRITEs must refer to the file number, not to the name. After the name and address are entered at the terminal, it is written on the file (WRITE 1, POS: NA,ST,CS;). The first parameter, 1, is the file number; the second parameter, POS, is the current logical record. It starts with 1 and increments by units with each person. "NA" is the variable containing the name, "ST" the street address and "CS" the city/state portion of the address. The program will loop indefinitely until the user is finished and hits RETURN to "NAME? ". The data is now in the file.

7. Write a program to print the file contents in specified format. Because labels are to be printed, the name, street address, and city and state are on three separate lines, with three blank lines between each person.

```
10   ASSIGN "MAIL",1; EFILE 1,1;
20   POS=1;
30   "START";
40   READ 1, POS: NA,ST,CS; IF EOF 1 [ END ] ;
50   PRINT NA; PRINT ST; PRINT CS; LIN 3; POS=POS + 1;
60   GOTO "START";
```

The program must again assign the file "MAIL" to a file number (ASSIGN "MAIL", 1;). Then the end-of-file flag is set (EFILE 1,1). The first parameter here, 1, determines the file number; the second parameter, 1, if given a non-zero value, tells OPUS that if an end-of-file is reached (that is, there is no more data in the file), no error message should be printed -- instead, look for an EOF statement to determine what to do next. POS is the variable that determines which logical record is to be accessed. The program then reads the record in "MAIL" determined by the logical record pointer, POS. The first item read in the record is assigned to the variable, NA, the second to the variable, ST, and the third to CS. If there is no data in the record (end-of-file), EOF will be set to 1 and the program will terminate. However, assuming that there is data, the program will print on the terminal the name and address of the person, increment POS by 1 (to get to the next logical record), and return to read the next record. The program loops until all records have been read and the data written on the terminal.

OPEN

This command is used to create a disc file.

Format: OPEN $expr_1$, $expr_2$, $expr_3$<,$expr_4$>

Expression 1 is the name of the file -- any string of ASCII characters not exceeding 7 characters in length. Expression 2 is the number of logical records per sector. It must be an integer from 1 to 128 (the maximum number of bytes in a sector). The length of the logical record in bytes must be predetermined by the user according to data that must go into one logical record. The number of logical records per sector may then be calculated by dividing the number of bytes in a sector by the number of bytes in a logical record. Expression 3 is the maximum number of logical records necessary in the file. Its value must be an integer from 1 to 65535.

The command enters the name of the file into the disc directory and creates the file on the disc. OPUS does not immediately allocate all sectors needed to hold the maximum number of logical records. Instead, as data is written into logical records, empty sectors will be retrieved as necessary. Therefore, one may open a file that potentially will require 1000 sectors, but none of these sectors will be reserved for the file until data is actually written into them. Example:

```
OPEN "SCRATCH", 1, 700;
```

Expression 4 defines the disc drive. This may be one of the following:

a. drive number from 0 to n, where n + 1 is the number of drives on the system, or

b. disc tag (an ASCII character string identified with the diskette during system generation). If the diskette is not present, or the drive containing the diskette was not previously enabled, OPUS will enter the Disc Swap Routine.

If expression 4 was not entered, OPUS will assume that the diskette is in the drive last declared in the DISC command.

ASSIGN

This command will retrieve a specified file from the disc and assign it a file number.

Format: ASSIGN $expr_1$, $expr_2$ <$expr_3$>

Expression 1 must be the name of a disc file that was previously created on the disc, and expression 2 must be a valid file number, i.e., an integer from 1 to the maximum number, as declared in system initialization. A file must be ASSIGN-ed before data may be read from, or written on, the file. At any point during a program, a different file may be assigned to the same file number. Only one file may be assigned in one ASSIGN statement.

Expression 3 defines the disc on which the file resides. This may be entered as one of the following:

a. drive number from Ø to n, where n + 1 is the number of drives on the system, or

b. disc tag (an ASCII character string identified with the diskette during the disc format procedure in the System Generation Routine). If the diskette is not present, or the drive containing the diskette was not previously enabled, OPUS will enter the Disc Swap Routine.

If expression 3 is not entered, OPUS will assume that the file resides on the drive last declared in the DISC command.

```
10   ASSIGN "XFILE", 1;
20   INPUT "WHAT DO YOU WISH TO WRITE? ", N;
30   WRITE 1, 1: N;
40   PRINT N, "HAS BEEN WRITTEN INTO THE FIRST RECORD OF XFILE.";
50   PRINT "IT READS BACK AS: ";
60   READ 1, 1: N; PRINT N;
                .
                .
                .
WHAT DO YOU WISH TO WRITE?  THIS IS A TEST
THIS IS A TEST   HAS BEEN WRITTEN INTO THE FIRST RECORD OF XFILE.
IT READS BACK AS:
THIS IS A TEST
```

READ

The READ statement must be used to read data from a disc file.

Format: READ $expr_1$, $expr_2$: variable list

Expression 1 is the required file number -- the number to which the file was previously ASSIGNed. If no file was assigned to this number, an error will occur. Expression 2 is the logical record to be read within the file. The value must be a positive integer greater than zero and less than or equal to the maximum number of logical records as declared in the OPEN command. The variable list is a list of variable names that are to be assigned the values of data that are in the logical record. The data in the logical record will be read sequentially and assigned to the variables from left to right. If a variable in the list is a matrix variable, data in the record will be assigned sequentially to each matrix element. There must be at least as many data items in the record as there are variables and/or matrix elements in the list, or an end-of-record (EOF/EOR) error will occur. If there are more data items in the logical record than in the variable list, the remaining data items will be ignored. The READ statement always starts at the beginning of the designated logical record and will never read past the end of it. It is not possible to read data sequentially from a file, disregarding logical record boundaries. The logical record MUST be specified.

```
10   ASSIGN "XFILE", 1;
20   READ 1, 1: A;
30   PRINT A;
          .
          .
          .
THIS IS A TEST
```

WRITE

The WRITE statement is used to write data into a disc file.

Format: WRITE $expr_1$, $expr_2$: list

Expression 1 is the file number to which the requested file was previously ASSIGNed. All files must first be assigned a file number before any reference (READ or WRITE) can be made to them. Expression 2 is the logical record that is to be written. This number must be an integer greater than zero and less than or equal to the maximum number of logical records in the file, as initially declared in the OPEN statement. The list consists of one or more variables, constants, or expression values that are to be written in the logical record. If a matrix variable is listed, the entire contents of the matrix will be written on the record.

An entire logical record must be written at once. It is not possible to change one portion of the record, leaving other values unchanged. One must first READ the entire contents of the record, change the appropriate variable, and then WRITE all the variables back on the record. It is not possible to write data across logical record boundaries. If one attempts to write more values on a record than will fit, an end-of-record (EOF/EOR) error will occur, unless suppressed by the EFILE command.

```
10   ASSIGN "XFILE", 1;
20   INPUT "VALUE? ", A;
30   WRITE 1, 1: A;
                .
                .
                .
VALUE? THIS IS A TEST
```

CLOSE

This command will close a file that was previously assigned in the ASSIGN statement. The disc will be updated and the file released.

Format: CLOSE <expr>

The expression is a file number to which the file was previously assigned.
Example:

```
10   ASSIGN "XFILE", 1;
20   READ 1, 1: A;
30   PRINT A;
40   CLOSE 1;
```

Simply using CLOSE with no designated file will release all files and update the disc. It is good practice to use a CLOSE statement at the end of any program which changes data on the disc, to assure that the disc has indeed been updated.

```
40   CLOSE;
```

PURGE

This command will delete all data from a specified disc file logical record.

Format: PURGE $expr_1$, $expr_2$

Expression 1 is a valid file number to which a file was previously assigned. Expression 2 is the number (an integer from 1 to the maximum number in the file) of the logical record that is to be purged from the file.

FILE

This function will return certain information regarding a disc file.

Format: FILE $expr_1$, $expr_2$

Expression 1 is the file position number -- the number to which a file has previously been assigned, within the program. Expression 2 is an integer from 1 to 6, whose value will cause the function to return the following:

1 Name of the file assigned to this position
2 The number of bytes per logical record
3 The number of logical records per sector, as established at the time of file creation
4 The maximum number of logical records allowed in the file
5 The number of logical records that have had data written in them
6 The number of logical records that are empty (have had no data written in them)

```
10   ASSIGN "XFILE", 1;
20   PRINT "THE NAME OF THE FILE IS:", FILE ( 1, 1) ;
30   PRINT "NUMBER OF BYTES PER LOGICAL RECORD:", FILE ( 1, 2) ;
40   PRINT "NUMBER OF LOGICAL RECORDS PER SECTOR:", FILE ( 1, 3) ;
50   PRINT "MAXIMUM LOGICAL RECORDS:", FILE ( 1, 4) ;
60   PRINT "LOGICAL RECORDS USED:", FILE ( 1, 5) ;
70   PRINT "LOGICAL RECORDS EMPTY:", FILE ( 1, 6) ;
                          .
                          .
                          .
THE NAME OF THE FILE IS:   XFILE
NUMBER OF BYTES PER LOGICAL RECORD:  128
NUMBER OF LOGICAL RECORDS PER SECTOR:   1
MAXIMUM LOGICAL RECORDS:   1
LOGICAL RECORDS USED:   1
LOGICAL RECORDS EMPTY:   0
```

KILL

This command is used to delete (purge) either a program or a data file from the disc.

Format: KILL $expr_1$ <,$expr_2$ >

Expression 1 must be a valid name of a program or file currently on the disc. The name should be followed by a backslash (\) and one of the following type codes:

S = Source program
O = Object program
F = Data file
D = OPUS/TWO dimensioned data file

If the type code is not entered, it is assumed to be a source program.

Expression 2 defines the disc drive, if the device specified is the disc. This may be one of the following:

a. drive number from Ø to n, where n + 1 is the number of drives on the system, or

b. disc tag (an ASCII character string identified with the diskette during the disc format procedure in the System Generation Routine).

If the diskette is not present, or the drive containing the diskette was not previously enabled, OPUS will enter the Disc Swap Routine. If expression 2 was not entered, OPUS will assume the diskette is in the drive last declared in the DISC command. Examples:

```
KILL "TEST\O"
KILL "FILE\F",1
KILL "SOURCE"
```

A program or file is erased from the disc by this operation. KILL will not affect any program that may currently be in memory. If an updated version of a program is to replace an old version on the disc, the latter must first be KILLed and the new version then SAVEd or CSAVEd.

End Of File: EOF

This function will be executed only if the EFILE command has been declared TRUE for the appropriate file. If so, this statement will determine whether or not an end-of-file or end-of-record has been reached by the last READ/WRITE statement affecting the file.

Format: EOF expr

The expression must be a valid file number, corresponding to the same file declared under the EFILE statement. The function will return a TRUE value (1) if an end-of-file was reached in the last READ or WRITE command. It will return a FALSE value (Ø) if no end-of-file was reached, or no READ or WRITE statement was executed. See End of FILE (the EFILE command).

```
10   ASSIGN "XFILE", 1;
20   EFILE1, 1;
30   INPUT "RECORD #? ", R;
40   READ 1, R: A;
50   IF EOF 1[PRINT "NO SUCH RECORD"; END];
60   PRINT A;
          .
          .
          .
RECORD #? 5
NO SUCH RECORD
FINE
```

End of FILE: EFILE

This command determines whether or not an end-of-file/end-of-record reached by a READ/WRITE/PURGE command is under program control, i.e., may or may not be suppressed.

Format: EFILE $expr_1$, $expr_2$

Expression 1 must be a valid file number, an integer from 1 to the maximum number declared in system generation, declaring which file is to be affected. Expression 2 must have either of the following two values:

TRUE (non-zero): After this command has been executed, if an end-of-file is encountered during a READ/WRITE operation, OPUS will not print the normal end-of-file error message and terminate the program. Program execution will continue as if no end-of-file has been reached.

FALSE (zero) : If an end-of-file is encountered, OPUS will print the appropriate error message and terminate the program. Unless this operation was previously executed with a TRUE operand, end-of-record/end-of-file errors will terminate the program.

By itself, this command will do nothing but suppress program termination when an end-of-file is reached. Generally, it will be used in conjunction with the EOF statement. Note that there is no way to determine whether an end-of-record, as opposed to an end-of-file, was reached.

```
 10  ASSIGN "XFILE", 1;
 20  PRINT "THE FILE NAMED 'XFILE' SHOULD HAVE BEEN OPENED TO ONE";
 30  PRINT "LOGICAL RECORD.  IF WE TRY TO READ THE SECOND, WE WILL";
 40  PRINT "GET AN 'EOF/EOR' ERROR.  HOWEVER, IF 'EFILE' IS FIRST SET,";
 50  PRINT "THEN THE PROGRAM WILL CONTINUE EXECUTION.";
 60  PRINT ;
 70  EFILE 1, 1;
 80  READ 1, 2: N;
 90  IF EOF ( 1)
100  [ PRINT "THERE WAS NO RECORD 2, BUT EXECUTION CONTINUED."; ] ;
          .
          .
          .
THE FILE NAMED 'XFILE' SHOULD HAVE BEEN OPENED TO ONE LOGICAL RECORD.  IF
WE TRY TO READ THE SECOND, WE WILL GET AN 'EOF/EOR' ERROR.  HOWEVER, IF
'EFILE' IS FIRST SET, THEN THE PROGRAM WILL CONTINUE EXECUTION.

THERE WAS NO RECORD 2, BUT EXECUTION CONTINUED.
```

DISC

This command will enable a particular disc, declaring this as the default disc in instances where a specific disc is not given with other disc commands.

Format: DISC expr

The expression must be the drive number from Ø to n, where n + 1 is the number of drives on the system, or the disc TAG which was entered when the diskette was formatted. In the latter case, the disc must have been previously enabled or OPUS will go into the Disc Swap Routine.

Note: It is not necessary to use the DISC command before accessing a disc if one specifies the disc number with a disc command (GET, SAVE, OPEN, ASSIGN, etc.); the DISC command simply sets up a default value. If the DISC command has not been executed and another command is given, assuming a default disc, the error message "-NO DISC DECLARED-" will occur.

LIBrary

The LIBrary command will print out all programs and files residing on a disc.

Format: LIB $expr_1$: $expr_2$

Expression 1 is an optional device number to which the directory will be printed. If not specified, the current terminal will display the directory. Expression 2 is either a valid disc drive number from Ø to n, where n + 1 is the number of drives on the system, or a disc tag. If not specified, the disc is assumed to by the one last declared in the DISC command. For each program, LIBrary lists the name of the program, whether it is a source or object program, and the length in number of bytes. For each file, LIBrary lists the name of the file, the number of logical records per sector, the maximum number of logical records allowed.

NAME	TYPE	LENGTH	LR/SECTOR	MAX LR
TESTF	F		2	1ØØ
FIXIT	S	2345		
HELP	O	346		
DFILE	D		3	ØØ1436

NAME : Name of the program or file
TYPE : F = Data file
S = Source program
O = Object program
D = Dimensioned data file (OPUS/TWO)
LENGTH : The length (in bytes) of a program
LR/SECTOR: Number of logical records in one sector
MAX LR : Maximum number of logical records allowed in the file.

IX. BRANCH AND BLOCK OPERATIONS

A. UNCONDITIONAL BRANCHING

There are two statements in OPUS that will unconditionally send program execution to a different part of the program, the GOTO and the GOSUB commands.

Labels

The section of the program which is to pick up execution is identified by means of a 'label'. The label is simply a string constant of any value and length that is located at the starting location of the program section. It may or may not be connected with a certain operation.

These commands require, as a parameter, a value that is equivalent to the label value. The characters, and the length of this command parameter, must be identical to the string constant label to which execution is to be transferred. The parameter may actually be a variable or other expression, but the value is what is considered.

When one of these statements is encountered, OPUS will take the label value and scan from the start of the program for a matching string constant. All statements requiring a label operand will be skipped. The first matching string it encounters is considered to be the location of the label constant. All block boundaries are ignored; that is, it is possible to jump into the middle of a block.

Because the object code is stored in Postfix notation and the operands precede the operators, a label constant may be part of another operation. That operation will be executed upon receiving control.

The following example shows the easiest method of program transfer. The label is "XYZ123".

```
 50  GOTO "XYZ123";
       .
       .
       .
400  "XYZ123"; PRINT "HERE I AM";
```

In the following example, the two strings, X & Y, are concatenated to form the label value for the GOTO statement. After execution of the GOTO, the program will print "THIS IS A LABEL".

```
 10  X="THIS IS "; Y="A LABEL";
 20  GOTO X&Y:
       .
       .
       .
100  PRINT "THIS IS A LABEL";
```

The programmer should be forewarned that using the GOTO to jump out of the middle of blocks may cause problems. See Section XV. for the details of such problems.

Subroutines

The GOSUB statement is an unconditional jump to a subroutine block. After executing the subroutine, control will be transferred back to the statement following the GOSUB. Program code is normally put in a subroutine when it is necessary to repeatedly execute that code in several places within the main body of the program. The normal format of subroutine usage is as follows:

Main program:

```
 10  . . .
 20  GOSUB LABEL              This line sends control to the subroutine.
 30  . . .
 40  . . .
 50  GOSUB LABEL              Another call is made to the subroutine.
       .
       .
       .
```

Subroutine:

```
100  LABEL (of subroutine)    The subroutine must have the label as the
110  . . .                    heading of the subroutine.
       .
       .
       .
200  RETURN                   A RETURN statement sends control back to the
                              operation following the GOSUB statement.
```

There is no limit to the number of calls that may be made to the subroutine. Code within a subroutine may call another subroutine. Subroutines may be nested as deeply as necessary, dependent only upon the amount of memory in the system.

Subroutines are normally placed at the end of programs. A subroutine should never be executed without going through a GOSUB statement. If it is, a "STACK OVERFLOW" error will occur.

All variables used outside and inside the subroutine are global; that is, their values will be the same wherever they are used. No local variables may be declared.

OPUS/TWO and OPUS/THREE have more extended subroutine capabilities with local variables (see Section XIV.).

GOTO

This command will unconditionally force program execution to the location determined by the argument.

Format: GOTO expr

The expression may be any number, string, or variable whose literal string value appears at the location at which the program should pick up execution. This literal string will be called a label of the routine. OPUS will scan the program, from the beginning, until a matching label is found. An error will result if no matching label is found. If more than one label exists, the first one will be assumed. All matching literal strings used by other GOTOs, GOSUBs, SCANs and EXTs will be ignored.

If the programmer keeps in mind that operands precede their operators in the object program, GOTOs may refer to a label that is an operand for a PRINT, INPUT or other statement. NOTE: GOTOs cannot refer to a line number, since all line numbers are deleted when a program is compiled.

```
10  PRINT "THIS IS AN INFINITE LOOP...";
20  PRINT "BREAK BY HITTING CONTROL + C...";
30  "START"; GOTO "START";
          .
          .
          .
THIS IS AN INFINITE LOOP...
BREAK BY HITTING CONTROL + C...
```

GOSUB...RETURN

A GOSUB causes the program to jump to the specified subroutine, and when a RETURN statement is reached within the subroutine, to return to the statement following the initial GOSUB.

Format: GOSUB expr

The expression is the label of the subroutine to which control is to be transferred. It may be a constant, variable, or expression. The value of the expression, in literal string format, must be the first item in the subroutine that is being called. An error and program termination will result if there is no such label. If the user keeps in mind that operands always precede their operators in the object program, GOSUBs may refer to a label that is the operand of the first statement in the subroutine. NOTE: GOSUBs cannot refer to a line number, since all line numbers are deleted when a program is compiled.

```
10   GOSUB "ROUTINE";
20   PRINT "NOW BACK TO MAIN ROUTINE...";
30   END ;
40   "ROUTINE"; PRINT " NOW EXECUTING SUBROUTINE...";
50   RETURN ;
        .
        .
        .
NOW EXECUTING SUBROUTINE...
NOW BACK TO MAIN ROUTINE...
```

RETURN

The RETURN statement must be used at the end of a subroutine to force program control back to the main routine.

Format: RETURN

There are no parameters. When the program counter reaches this statement, it will return to the statement directly following the GOSUB that initially called the subroutine. An error will occur if a RETURN statement is attempted and there was no prior GOSUB.

B. CONDITIONAL BLOCK OPERATIONS

The statements included in this section will execute blocks of code if certain conditions are satisfied.

A block of code is defined as a section of program code delimited by a block initiator and a block terminator. The operations using this block format are as follows:

Initiator	Terminator
[	]
LOOP	NEXT
WHILE	CONTinue

The bracket block is explained at length in Section II.

The LOOP and WHILE blocks of code are conditionally executed only if the parameter of the block initiator results in a TRUE condition.

Upon executing a block initiator, OPUS will put this location and any other pertinent information in the function stack. Execution of the block terminator will remove this data from the stack. With this in mind, it should be apparent that attempts to jump out of or into the middle of a block may result in errors. If a block is exited without executing the block terminator, OPUS assumes the following code is still in that block. Thus blocks become needlessly nested until a stack overflow error occurs. If a block terminator is executed without a previous block initiator, a stack overflow will also occur, as OPUS pops everything off the stack looking for the initiator.

Blocks may be nested within each other as deeply as memory allows.

Subroutines may also be considered blocks of code with the GOSUB as the block initiator and the RETURN as the block terminator.

There are three statements that conditionally execute a bracket block of code:

IF
ELSE
ON

In the first two statements, a TRUE/FALSE condition must exist, determining whether the following block is to be executed. The ON statement will execute a specific block of code, depending upon the value of the parameter.

IF

This conditional command may be used when certain program code is to be executed only if a specified condition is TRUE.

Format: IF expr block

If the expression returns a TRUE (non-zero) value, the block will be executed. If it returns a FALSE (Ø) value, the block will be skipped, and program execution will continue after the block. The code within the block must be inserted within brackets ([ ]) unless it consists only of one statement, one variable, or one constant. The ELSE statement may be used in conjunction with the IF statement to execute a block of code should the IF condition fail.

```
10  INPUT "NUMBER? ", N;
20  IF N# TRU ( N) [ PRINT "DECIMAL POINT MAY NOT BE USED"; ] ;
                  .
                  .
                  .
NUMBER? 1.2
DECIMAL POINT MAY NOT BE USED
```

IF...ELSE

The ELSE command may be used in conjunction with the IF statement.

Format: IF expr block$_1$ ELSE block$_2$

If the previous IF statement has failed, the ELSE statement will execute the succeeding block of code. Otherwise, the block will be skipped. The code within the block must be inserted within brackets ([ ]) unless it consists only of one statement, one variable or one constant. If there was no previous matching IF statement, a stack overflow error will occur.

If the expression is TRUE (non-zero value), block 1 will be executed and block 2 will be skipped. If the expression is FALSE (zero value), block 1 will be skipped and block 2 will be executed.

```
10  "START";
20  INPUT "TYPE A NUMBER: ", N;
30  IF NONE [ END ; ] ;
40  IF N >5 [ PRINT "YOUR NUMBER IS GREATER THAN 5";]
50  ELSE [ IF N <5AND N# 5 [ PRINT "YOUR NUMBER IS LESS THAN 5";]
60  ELSE [ PRINT  "YOUR NUMBER IS EQUAL TO 5"; ] ]  ;
70  GOTO "START";
                .
                .
                .
TYPE A NUMBER: 7
YOUR NUMBER IS GREATER THAN 5
TYPE A NUMBER: 3
YOUR NUMBER IS LESS THAN 5
TYPE A NUMBER:
FINE
```

LOOP...TO...NEXT

The LOOP statement allows the section of program code within a loop block to be continuously executed until the loop variable reaches a specified value, at which point program execution continues on after the loop block. The block initiator is the LOOP statement, and the block terminator the NEXT statement.

Format: LOOP variable, $expr_1$ TO $expr_2$...NEXT<$expr_3$>

The variable must be any valid simple variable name or variable matrix element name. The expression values should be any numerical value (strings are converted to numbers). When the program counter encounters the LOOP statement, the value of expression 1 is assigned to the variable. This is the starting value of the loop variable. The TO statement compares the value of the loop variable to expression 2. If the variable is less than or equal to the expression (numerical value), the program code within the loop is executed. However, if the variable is greater than expression 2, the program skips to the program code immediately following the corresponding NEXT statement.

When the loop is executed and a NEXT statement is reached, OPUS checks for the expression 3 value. If such a value exists, it is added to the loop variable; if there is no such parameter, the loop variable is incremented by one. The NEXT statement then causes program execution to return to the previous TO statement, which in turn checks to see if the loop variable has reached the maximum. The program loops in this fashion until the loop variable has exceeded expression 2; at this point, it skips to the statment following the NEXT statement.

Loops may be nested within each other. There must always be a corresponding NEXT for every LOOP. It is not recommended to jump out of loops. OPUS allocates a stack buffer for loop parameters. The parameters are not purged from the stack until a loop has completed its cycling normally. Therefore a jump out of a loop will leave these loop parameters in the stack. These may be picked up by other operations further down the line, causing unexpected results.

```
10   LOOP K, 1TO 5;
20   PF "SB", K;
30   NEXT ;
        .
        .
        .
1 2 3 4 5
```

```
10    REM   "LOOP WITH INCREMENT OF 5";
20    LOOP I, 1TO 5;
30    LOOP J, 1TO 30;
40    PF "SB", J;
50    NEXT 5; PRINT ;
60    NEXT ;
            .
            .
            .
1   6   11   16   21   26
1   6   11   16   21   26
1   6   11   16   21   26
1   6   11   16   21   26
1   6   11   16   21   26
```

WHILE...CONTinue

The WHILE statement is used to repeatedly execute a section of program code until a condition becomes FALSE. The WHILE statement is the block initiator, and the CONTinue the block terminator.

Format: WHILE expr; . . .CONT

The value of the expression is determined to be TRUE (numerical value not equal to Ø) or FALSE (numerical value equal to Ø). If TRUE, the program will continue executing sequentially until it hits the CONTinue statement, at which point, control will be transferred back to the WHILE statement and the expression re-evaluated. As long as the expression remains TRUE, this loop will be executed. However, once the expression becomes FALSE, the program will skip out of the loop to the statement following the CONTinue statement.

```
10   X= 10;
20   WHILE X>0; PF "SB", X;
30   X= X- 1; CONT ;
        .
        .
        .
10 9 8 7 6 5 4 3 2 1
```

ON

The ON statement is a conditional statement that will execute a specific block of program code, depending upon the value of the argument.

Format: ON expr $block_1$\$block_2$\. . .\$block_n$

If the expression has the value of 1, block 1 will be executed, and following blocks in the operation will be ignored. If it has a value of 2, only block 2 will be executed; all others will be ignored. If the expression has the value n, only the nth block will be executed. The code within each block must be inserted within brackets ([ ]) unless it consists of only one statement, one variable, or one constant. The same logic holds with the following format:

Format: ON expr, statement $operand_1$\$operand_2$\. . .\$operand_n$

If the expression has a value of n, for instance, the statement will use the nth operand and ignore all other operands.

The expression must have an integer value greater than zero, or an error will occur.

```
 5   "START";
10   INPUT "TYPE A NUMBER: ", N;
20   ON N[ PRINT "ONE" ] \ [PRINT "TWO"] \ [PRINT "THREE"] ;
30   GOTO "START";
                .
                .
                .
TYPE A NUMBER : 1
ONE
TYPE A NUMBER: 2
TWO
TYPE A NUMBER: 3
THREE
```

NOTE: Line 20 may also be written as:

```
20   ON N, PRINT "ONE" \"TWO" \ "THREE":
```

X. BOOLEAN & RELATIONAL OPERATIONS

A. BOOLEAN OPERATIONS

Boolean operators will treat the required operand values as logical numbers. Operands will first be converted to number format. If the numerical value is non-zero, the operand is TRUE. If the value is zero, the operand is FALSE. Given this TRUE/FALSE condition of the operands, the Boolean operator will return a TRUE (1) or FALSE (Ø) value, depending upon the operation. The Boolean operators in OPUS are:

Binary:	AND	Returns TRUE if both operands are TRUE.
	OR	Returns TRUE if either operand is TRUE.
Unary :	NOT	Returns TRUE if the operand is FALSE.

FALSE

FALSE is one of two logical values of any number or string. An operand value will be considered FALSE if, and only if, its numerical value is equal to zero. A string such as "HORSE" is FALSE because its numerical value is Ø. However, "34" is TRUE because its numerical value is 34 (non-zero).

TRUE

TRUE is a logical value of an expression. If an expression has a non-zero numerical value, it is considered to be TRUE, whereas a zero numerical value will be FALSE. Thus numbers such as 34, -.44, and 109.4 are TRUE; strings such as "35", "-5", and "4DDD" are TRUE because their numerical forms are non-zero. However, "HORSE" is FALSE because conversion to a number results in Ø. Some statements will return a TRUE or FALSE value to the operand table, depending upon the condition met. A TRUE value will always be returned as 1, a FALSE value as Ø.

AND

This Boolean binary operator may be used to determine whether or not two expressions are TRUE.

Format: $expr_1$ AND $expr_2$

If the numerical value of expression 1 is TRUE (non-zero) and the numerical value of expression 2 is TRUE (non-zero), the value returned will be TRUE (1). If either expression is FALSE (Ø), the value returned will be FALSE (Ø).

```
10   INPUT "FIRST NUMBER? ", A, " SECOND? ", B;
20   PRINT A, "AND", B, "=", AAND B;
                    .
                    .
                    .
FIRST NUMBER? 1  SECOND? 0
1   AND   0   =   0
```

OR

This is a Boolean binary operator that determines whether or not one of two operands has a TRUE value (non-zero).

Format: $expr_1$ OR $expr_2$

If either expression 1 or expression 2 has a non-zero numerical value, this operation will return a TRUE value (1). If both expressions have a FALSE value (Ø), the operation will return a FALSE value.

```
10   INPUT "FIRST NUMBER? ", A, "  SECOND? ", B;
20   PRINT A, "OR", B, "IS", AOR B;
                    .
                    .
                    .
FIRST NUMBER? 1  SECOND? 0
1   OR   0   IS   1
```

NOT

The NOT function logically negates its argument.

Format: NOT expr

If the expression has a numerical value of zero, the NOT function will return a TRUE value (1). If the expression has a non-zero value a FALSE value (Ø) will be returned.

```
5    "START";
10   INPUT "TYPE A NUMBER: ", N;
20   IF NOT N[ PRINT "THAT WAS ZERO";]  ;
30   GOTO "START";
          .
          .
          .
TYPE A NUMBER: 1
TYPE A NUMBER: 0
THAT WAS ZERO
```

B. RELATIONAL OPERATIONS

Relational operators compare the values of two operands and return a TRUE or FALSE value, depending upon the relationship of the two. A relational operation will always be a binary operation.

Format: $expr_1$ operator $expr_2$

The following operators are available in OPUS to compare one value with another:

<	The first expression is less than the second
>	The first expression is greater than the second
#	The first expression is not equivalent to the second
IS	The first expression is equivalent to the second
<=	The first expression is less than or equal to the first
>=	The first expression is greater than or equal to the first

The expression values may be in either string or number format, but before the compare operation is actually executed, OPUS will make the appropriate conversions to make sure they are in the same format. The following rules apply to the conversions and comparisons:

1. If both of the values are in number format, the numerical value of the first is compared to the numerical value of the second.

2. If both of the values are in string format, the ASCII code representation of each character in the first value is compared numerically (from left to right) to the corresponding character in the second value. Refer to the ASCII Table for the code representation of each character.

3. If one expression value is in string format and the other is in number format, the one in number format will first be converted to a string. At this point, the string with the least number of characters will be given enough leading zeros to make it the same length as the other. The two strings will then be compared according to Rule 2. above. Note that decimal points will not be lined up. Should the user specifically want a numerical comparison and the format of the values is in doubt, the NUMber function should be used.

Less Than: $<$

This symbol designates the relational binary operation of "less than".

Format: $expr_1 < expr_2$

The value returned from the operation will be TRUE (1) only if the value of expression 1 is less than the value of expression 2. If the value of the first is greater than or equal to the second, the value returned will be FALSE (Ø). OPUS will make sure that the value formats of the two expressions are the same prior to this operation.

```
PRINT 2<3
1
```

Greater Than: >

This symbol designates the relational binary operation for "greater than".

Format: $expr_1 > expr_2$

If the value of expression 1 is greater than the value of expression 2, the operation will return a TRUE value (1). If the former is less than or equal to the latter, the returned value will be FALSE(Ø). OPUS will make sure that the value formats of the two expressions are the same prior to this operation.

```
A=10; IF A>5[PRINT "TRUE"];
TRUE
```

Not Equal: #

The octothorpe (pound sign) is a relational binary operator that determines the equivalency of the two operands.

Format: $expr_1$ # $expr_2$

If the two expressions are not equal, the operation will return a TRUE value (1). If they are the same, the operation will return a FALSE value (Ø). OPUS/ONE will make sure that the value formats of the two expressions are the same prior to this operation.

```
10   INPUT "NUMBER? ", N;
20   IF N# 3 [ PRINT "NO GOOD. . .TRY AGAIN";GOTO "NUMBER? " ] ;
                    .
                    .
                    .
NUMBER? 2
NO GOOD. . .TRY AGAIN
NUMBER? 3
```

IS

This relational binary operator scans the two operands and determines whether or not they are equivalent.

Format: $expr_1$ IS $expr_2$

This IS operator will return a TRUE value (1) if the two expression values are the same. It will return a FALSE value (Ø) if they are different. For instance:

```
3.4 IS 3.4     Returns a TRUE value
3.4 IS "3.4"   Returns a TRUE value
"45" IS 45     Returns a TRUE value
Ø IS "DOG"     Returns a FALSE value
```

```
10   INPUT ":", A;
20   IF AIS "CAT"  [PRINT "TRUE"; ] ;
                   .
                   .
                   .
:CAT
TRUE
```

Less Than or Equal To: <=

This symbol designates the relational binary operation of "less than or equal to".

Format: $expr_1$ <= $expr_2$

The value returned from the operation will be TRUE (1) only if the value of expression 1 is less than or equal to the value of expression 2. If the value of the first is greater than the second, the value returned will be FALSE (Ø). OPUS will make sure that the value formats of the two expressions are the same prior to this operation.

```
PRINT 3 <=3;
1
```

Greater Than or Equal To: >=

This symbol designates the relational binary operation for "greater than or equal to".

Format: $expr_1$ >= $expr_2$

If the value of expression 1 is greater than or equal to the value of expression 2, the operation will return a TRUE value (1). If the former is less than the latter, the returned value will be FALSE (Ø). OPUS will make sure that the value formats of the two expressions are the same prior to this operation.

```
A=5; IF A>=5[PRINT "TRUE"];
TRUE
```

XI. FUNCTIONS

A function is a statement that operates on some given parameter(s), and then optionally returns a result to the operand table for use by another statement. The following functions are available in OPUS:

ABS	Returns the absolute value of the operand
ASC	Returns the ASCII character representation of the operand
ATN	Returns the arctangent of the parameter
BRK	Suppresses program interrupts
COS	Returns the cosine of the parameter
DATE	Returns the day, month, or year, depending upon the parameter
EXP	Returns the value of e^x, where x is the parameter
FETCH	Returns the contents of a specified memory core location
LEN	Returns the length (characters or digits) of the parameter
LOG	Returns the natural logarithm of the parameter
MAX	Returns the greater of two numerical values
MIN	Returns the lesser of two numerical values
NONE	Returns a value specifying whether or not there was data entered in the last INPUT statement
NUM	Forces the argument into number form and returns that value
RND	Returns a random number between Ø and 1
SGN	Returns the sign (negative, zero, positive) of the argument
SIN	Returns the sine of the argument
SQR	Returns the square root of the argument
STR	Forces the argument into string format and returns that value
STUFF	Changes a byte in memory
TAN	Returns the tangent of the argument
TRU	Truncates the argument into integer form

A. EXPONENTIAL & LOGARITHMIC FUNCTIONS

The following exponential and logarithmic functions are available in OPUS:

EXP Returns the value of the Euler constant, e, raised to the power, x, where x is the value of the argument
LOG Returns the natural logarithm of the argument

No more than 6 digits may be considered accurate on all calculated values.

Each function is a unary operator; that is, one operand is required for execution, and the resulting value is returned for use by another statement.

EXPonent

The EXPonential function returns the exponential constant, e (2.71828...), raised to the power of the argument.

Format: EXP expr

The expression may be any negative or positive rational number.

```
PRINT EXP 4;
54.5982
```

LOGarithm

The LOGarithm function will return the natural logarithm of the argument.

Format: LOG expr

The expression must have a numerical value greater than zero.

```
PRINT LOG 2
.301029
```

B. TRIGONOMETRIC FUNCTIONS

The following trigonometric functions are available in OPUS:

SIN	To determine the sine of an angle
COS	To determine the cosine of an angle
TAN	To determine the tangent of an angle
ATN	To determine the arctangent of the argument

No more than 6 digits may be considered accurate on all calculated values.

SINe

This trigonometric function will return the sine of its argument.

Format: SIN expr

The expression is assumed to be a value expressed in radians. The SINe function returns the sine of this angle, from -1 to +1 (inclusive).

```
PRINT SIN 2
-.909297
```

COSine

This trigonometric function will return the cosine of an argument.

Format: COS expr

The expression returns an argument which is assumed to be in radians. The value returned by the function will be between -1 and 1 (inclusive).

```
PRINT COS 2
-.41646
```

TANgent

This trigonometric function determines the tangent of the given argument.

Format: TAN expr

The expression is a numerical value assumed to be in radians, and this function returns its tangent.

```
PRINT TAN 2
-2.18503
```

ArcTaNgent

This trigonometric function is used to find the arctangent in radians, of an expression.

Format: ATN expr

The expression may have any numerical value. The value returned (in radians) will have a value between $-\pi/2$ and $\pi/2$.

```
PRINT ATN -2.18503
-1.14159
```

C. GENERAL FUNCTIONS

ABSolute

The absolute value function is a unary operator.

Format: ABS expr

If the numerical value of the expression is negative, this operation will return the corresponding positive value. If the numerical value is zero or is positive, there will be no change, and the same value will be returned.

```
10   INPUT "NUMBER? ", N;
20   PRINT "THE ABSOLUTE VALUE OF", N, "IS", ABS ( N) ;
                                    .
                                    .
                                    .
NUMBER? -4.2
THE ABSOLUTE VALUE OF    -4.2   IS   4.2
```

ASCII

This function will determine the ASCII character of the corresponding decimal number code.

Format: ASC expr

The expression must have a value between Ø and 255 (inclusive), corresponding to the decimal representation of an ASCII character of this number. For example:

```
PRINT ASC 7
```

will generate a bell (control G) on terminals with the bell capability.

```
10   INPUT "DECIMAL EQUIVALENT? ", N;
20   PRINT "THE ASCII CODE WHICH CORRESPONDS TO", N, "IS", ASC ( N) ;
                    .
                    .
                    .
DECIMAL EQUIVALENT? 65
THE ASCII CODE WHICH CORRESPONDS TO  65  IS  A
```

BReaK

The BReaK function is used to disable or enable terminal interrupts during program execution.

Format: BRK expr

If the numerical value of the expression is zero, interrupts are enabled, i.e., by pressing any key during output or computation, program execution will be terminated. Unless previously set to a non-zero value, interrupts will automatically be enabled. If the numerical value of the expression is not zero, the user will be "locked out", i.e., nothing will interrupt the program except a machine HALT or normal termination.

Note that the terminal must have had an interrupt routine specified during the System Generation Routine in order for interrupts to work. If not, this function and Control C have no meaning, because interrupts are not possible in any situation.

```
10   PRINT "WHEN THIS LINE IS COMPLETED, YOU WILL NOT BE ABLE TO ";
20   PRINT "BREAK THE PROGRAM EXECUTION.";
30   BRK ( 1 ) ;
40   "START";
50   LOOP K, 1TO 5;
60   ON X+ 1 [ PF "SB", "ENABLED";] \ [ PF "SB", "DISABLED"; ] ;
70   NEXT ;
80   IF XIS 1 [ END  ; ] ;
90   PRINT ;
100  PRINT "BREAK FUNCTION NOW DISABLED.";
110  BRK ( Ø ) ;
120  X= 1;
130  GOTO "START";
          .
          .
          .
WHEN THIS LINE IS COMPLETED, YOU WILL NOT BE ABLE TO
BREAK THE PROGRAM EXECUTION.
ENABLED ENABLED ENABLED ENABLED ENABLED
BREAK FUNCTION NOW DISABLED.
DISABLED DISABLED DISABLED DISABLED DISABLED
FINE
```

DATE

This function determines the current day, month and year.

Format: DATE expr

If the expression value is "1", the function returns the day, a number from 1 to 31: if the value is "2", it returns the month, a number from 1 to 12: and a value of "3" returns the year, a number from 00 to 99. Thus, if OPUS is up and running on November 25, 2001, the following values will be returned when using the date command:

```
DATE 1  Returns  25
DATE 2  Returns  11
DATE 3  Returns  01
```

Apparently the user must determine the century by himself.

Please NOTE: The DATE function is dependent on the date entered when OPUS is initially brought up. If the user enters the wrong date, OPUS has no way to pull the correct date out with this function.

```
10   PRINT "DURING INITIALIZATION, TODAY'S DATE WAS ENTERED";
20   PRINT "THE DATE TODAY IS:";
30   PRINT DATE ( 2) & "/"& DATE ( 1) & "/"& DATE ( 3) ;
          .
          .
          .
DURING INITIALIZATION, TODAY'S DATE WAS ENTERED
THE DATE TODAY IS:
10/25/76
```

FETCH

This function may be used to read a byte directly from memory.

Format: FETCH expr

The expression is the memory address expressed as a decimal integer from Ø to 65535. The function will return the contents of this specified location as a decimal integer from Ø to 255. If a memory location is referenced that does not exist within the bounds of the memory configuration, a 255 will be returned.

```
10  A= FETCH ( 1ØØØ) ;
20  PRINT A;
          .
          .
          .
128
```

LENgth

The LENgth function will return the length of an operand (number of characters or digits).

Format: LEN expr

If the expression has a string value, the LENgth function returns the number of characters in the string. If it has a number value, this function first converts it to a string and then returns the length. The sign of the number and the decimal point will be counted as characters. For example:

```
LEN "HORSE"      Returns a length of 5
LEN 1.34         Returns a length of 4
```

```
10  INPUT ":", A;
20  PRINT "THE LENGTH OF", A, "IS", LEN ( A) ;
              .
              .
              .
:10867
THE LENGTH OF   10867   IS   5
```

MAXimum

This binary operator will determine the greater numerical value of the two operands.

Format: $expr_1$ MAX $expr_2$

If the numerical value of expression 1 is greater than that of expression 2, MAXimum will return the value of expression 1. Otherwise, it will return the numerical value of expression 2.

```
10   INPUT "FIRST NUMBER? ", A, "  SECOND? ", B;
20   PRINT "THE LARGER IS:", AMAX B;
                  .
                  .
                  .
FIRST NUMBER? 1  SECOND? 2
THE LARGER IS:   2
```

MINimum

This binary operator will determine the lesser numerical value of the two operands.

Format: $expr_1$ MIN $expr_2$

If the numerical value of expression 1 is less than that of expression 2, MIN will return the value of expression 1. Otherwise, the value of expression 2 will be returned.

```
10   INPUT "FIRST NUMBER? ", A, "  SECOND? ", B;
20   PRINT "THE LESSER IS:", AMIN B;
                    .
                    .
                    .
FIRST NUMBER? 1  SECOND? 2
THE LESSER IS:   1
```

NONE

The NONE function may be used to determine whether or not the user entered data in an INPUT statement.

Format: NONE

If, during the previous INPUT statement, the user had hit the RETURN key (had entered no data), the NONE function would return a TRUE value (1). If the user had entered data, this function would return a FALSE value (Ø). Similarly, if an input device other than a user-controlled terminal (e.g., paper tape reader or cassette) is accessed, if only a carriage return is received to an INPUT statement, the NONE function will return a TRUE value; otherwise, it will return a FALSE value.

```
10   "START";
20   INPUT "TYPE SOMETHING ", A;
30   IF NONE [ END ] ELSE [ PRINT A]  ;
40   GOTO "START";
          .
          .
          .
TYPE SOMETHING HGHGHG
HGHGHG
```

NUMber

The NUMber function may be used to force a string value into number format:

Format: NUM expr

If the value of the expression is a string, this function will return the numerical value. If the expression value is already a number, the function will simply return this value.

Since OPUS automatically handles number-to-string and string-to number conversions, this function is rarely needed. The most important use, however, is insuring that data is correctly formatted when writing on a disc file. Files are created with a certain number of bytes per logical record. If a number in string format is written on the disc, it will usually take more bytes than the same number in number format. For example, "-23" requires 4 bytes and -23 requires 3 bytes. Such differences may throw off the initial calculations of the number of bytes needed in one logical record.

RaNDom

The RND function will produce a pseudo-random number.

Format: RND

There are no arguments for the function. The value returned will be a random number between Ø (inclusive) and 1 (exclusive). It will be carried out to the number of digits as declared for precision. OPUS determines the first random number (which will not always be the same) when the system is first brought up. New random numbers will be generated at each request, regardless of whether the same or different programs are being executed. The length of the sequence (number of random numbers that will be generated before the same sequence is repeated) is 2^{16}. It is not possible to request a repeat of a random number sequence.

```
10  A= RND ; PRINT A;
          .
          .
          .
.269515991
```

SiGN

The SiGN function determines the sign (negative, zero, or positive) of the argument.

Format: SGN expr

If the numerical value of the expression is less than zero, SGN will return -1. If the value is zero, a zero will be returned. If the value is greater than zero, the returned value will be 1.

```
10   INPUT "NUMBER? ", A;
20   PRINT SGN ( A) ;
          .
          .
          .
NUMBER? -3
-1
```

SQuare Root

This function determines the square root of a given argument:

Format: SQR expr

The expression must have a positive numeric value.

```
PRINT SQR 4
2
```

No more than 6 digits may be considered accurated.

STRing

This function forces the argument into string format.

Format: STR expr

If the value of the expression is a number, the STRing function will return this value as a string. If it was already a string, no change is made, and the expression value is returned.

STUFF

The STUFF command may be used to directly change a location in memory.

Format: STUFF $expr_1$, $expr_2$

Expression 1 is a valid memory address, an integer from Ø to 65535, specifying the particular location that is to be changed. Expression 2 is the value of the byte that is to replace the old byte at the memory location. This must be an integer from Ø to 255. After the command is executed, the byte in the memory location will have the new value. NOTE: Do not use this command casually anywhere in memory. It is very easy to wipe out the OPUS operating system, causing the computer to crash, should the wrong bytes be changed.

```
10   INPUT "ADDRESS? ", A, "  VALUE? ", B;
20   STUFF A, B;
                  .
                  .
                  .
ADDRESS? 16000  VALUE? 101
```

TRUncate

The TRUncate function may be used to truncate a number to the next least integer.

Format: TRU expr

If the numerical value of the expression is a fraction, the TRU function will return the integer value which is less than the fraction. If the value of an expression is already an integer, TRU will do nothing but return the same value. Some examples follow:

```
TRU  3.45          Returns 3
TRU  -2.888        Returns -3
TRU  "45"          Returns 45
TRU  .09           Returns Ø
TRU  "XX"          Returns Ø
TRU  -10           Returns -10

10   INPUT "NUMBER? ", A;
20   PRINT "TRUNCATED:", TRU A;
          .
          .
          .
NUMBER? 1.334
TRUNCATED:   1
```

XII. MATRICES

A. FUNDAMENTALS

Matrices are used for data storage and retrieval within a program. The matrix may be of any size that memory will hold, i.e., up through 255 dimensions, with as many elements in each dimension as necessary or possible. Each element may contain either a number or a string. All matrices must be initially dimensioned with the DIMension statement. This allocates a buffer in memory for the matrix. The DIM statement will cause all elements within the matrix to be set to zero. A matrix may be re-dimensioned within the program at any time. A DIM statement may appear anywhere in the program, but must precede any reference made to the matrix. Once a variable has been declared a matrix, it will remain a matrix throughout the program, though it may be re-dimensioned. If a variable has been used as a string or number variable, it may also be converted to a matrix by use of the DIM statement. In the following example, the matrix has been given 3 dimensions and will contain 60 elements:

```
DIM M(3,4,5)
```

And the next one has one dimension and 100 elements:

```
DIM CAT(100)
```

A particular matrix element must be referenced in the following manner:

Format: Variable ($expr_1$, $expr_2$, . . . $expr_n$)

The variable is any valid simple variable name previously declared in a DIM statement. Expression 1 is the element position of the first dimension, an integer greater than zero and less than or equal to the maximum number of elements as declared in the DIM statement. Expression 2 is the element position of the nth dimension, where n is the number of dimensions the DIM statement initially declared.

A matrix element may be assigned a value in the assignment statement; a matrix element reference may be used in any expression; a matrix element substring may be referenced in this manner:

Format: Matrix element $(expr_1,<expr_2>)

Expression 1 is the first character position of the substring. Expression 2 is the last character position of the substring. See Substrings for further explanation. Some examples of matrix element references follow. The first one assigns "HORSE" to the element:

```
M(1,1,2) = "HORSE"
```

In the next one, the substring, "OR", of the string, "HORSE", is printed:

```
PRINT M(1,1,2)$(2,3)
```

And finally, the matrix element is used in an expression; X is assigned the value of "HORSE CART":

```
X = M(1,1,2) & " CART"
```

There are no specific statements, other than the DIM statement, that operate on matrices only.

B. MATRIX OPERATIONS

DIMension

All matrices of any dimension must be dimensioned prior to any matrix operations. The DIMension command reserves a memory buffer to hold the matrix contents.

Format: DIM variable (expr$_1$<, expr$_2$ . . . expr$_n$>)

Where n = the number of expressions in the list.

The variable must be any valid variable name assigned to the matrix. Expression 1 is the number of elements allowed in the first dimension (row); expression 2 is the number in the second dimension (columns); expression n is the number in the nth dimension (n may not exceed 255). All expression values must be an integer greater than Ø. The DIM operation automatically sets all elements of the matrix equal to Ø. A matrix may be re-dimensioned at any point in a program with the DIM statement.

```
10   DIM A( 10, 10) ;
20   LOOP J, 1TO 10;
30   PRINT ;
40   LOOP K, 1TO 10;
50   PF "S,B5", A( J, K) ;
60   NEXT ; NEXT ;
          .
          .
          .
0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0
0    0    0    0    0    0    0    0    0    0
```

Matrix Element: !

This statement may be optionally used to specify a matrix element.

Format: matrix variable ! (list)

The matrix variable must be a valid variable that was previously DIMensioned. The list includes all element numbers defining the location within the matrix. Each element number must be an integer from 1 to the maximum, as declared in the DIM statement.

The use of this statement is optional. If not given, OPUS will automatically insert it during compilation. If a matrix error occurs (the variable has not been previously DIMensioned or one of the elements is out of range), an exclamation point (!) will be printed.

XIII. MISCELLANEOUS OPERATIONS

END

The END command may be used anywhere in a program to terminate program execution. When OPUS reaches END command, it automatically returns to command mode.

Format: END

An END command, however, is not necessary to stop program execution. When OPUS reaches the end of the program, it automatically returns to command mode.

REMark

The REMark statement may be inserted at any point in the source program to store comments.

Format: REM "comment"

The comment may be anything the programmer desires and must be enclosed within quotes. All REMark statements and the following string will be deleted in the object code. Statements on the same line will be executed.

```
10  REM "* * * THIS A REMARK * * *"
20  PRINT 1 + 2 / 3;
30  END;
                .
                .
                .
1.666666
```

SCAN

The SCAN statement is used to read data contained within the program.

Format: SCAN<label:><variable list or matrix variable>

The label determines the location of data and must be a literal string contained somewhere in the program. By specifying this label in the SCAN statement, a pointer is set to the label in the program code in preparation to read data. If no label is specified, it is assumed that the pointer was set by a previous SCAN statement. If not, an error will occur. The variable list is a list of variable names that receive the values read. The first variable will be assigned the first item, the second variable the second item, and so forth, until all variables have been assigned a value. The pointer to the data location will move forward as each variable is assigned a value.

Any section of program may be considered data. There are three types of data items: constants, variable names and statements. Constants will simply be assigned to the list variable as is. Variable names will first be converted to a string and then assigned to the list variable. Statements will be assigned to the list variable as a number. This number corresponds to the position of the statement as listed in the Statement Table (see Section XIII).

Should a label be included in the SCAN statement, the first variable read will be given the label value.

If the user just wants to read specific constants as data, these constants may simply be listed at the end of the program, separated by commas. It is wise to keep this data section out of program execution reach, because if executed, OPUS will try to use the data as operands for some statement it has yet to reach. This will most likely cause an overflow in the operand table.

```
10  SCAN "HERE": A,B,C; SCAN D,E;
20  PRINT "HERE", "I", "X";
30  END;
```

After execution of the above program, the following data will be read into the variables:

```
A will be "HERE"
B will be "I"
C will be "X"
D will be 28 (PRINT)
E will be 83 (;)
```

NOTE: Remember that the object program is stored in postfix notation -- that's why the PRINT is after the string values.

THEN

The THEN statement may be used whenever a parameter is indirectly implemented in an operation.

Format: THEN list

The list contains one or more expression values. The THEN statement does not in itself perform any operation on the parameters, but instead sets the parameters up for use in a different operation. In general, its use determines only the order of compilation, i.e., where the operand appears in relationship to the operators in the object program.

```
PRINT (IF X [THEN "X IS TRUE"]  ELSE  [THEN "X IS FALSE"] )
```

In this example, the THEN statement will set for the PRINT statement either "X IS TRUE" or "X IS FALSE", depending upon the value of X.

XIV. OPUS/TWO & OPUS/THREE SUPPLEMENT

In order to further enhance the capabilities of OPUS, several additional features have been added to OPUS/ONE. All OPUS/ONE programs are upward compatible with OPUS/TWO and OPUS/THREE.

Twenty-three new commands have been added, as well as additional file and disc handling capabilities.

A. OPUS/TWO

Error Trapping

All OPUS statement errors may be trapped by automatically sending program execution to a pre-defined subroutine. The subroutine may attempt to take corrective action after determining what type of error has occurred. Two statements are used to implement error trapping: ERRor and the "?".

ERRor

This command must be used to notify OPUS that error trapping is in effect, and to specify a subroutine to which all errors are to go.

Format: ERR expr

The expression is a label that is equivalent to the literal string at the beginning of the error subroutine. Upon hitting an error, OPUS will essentially execute a GOSUB to this subroutine. The user should terminate the subroutine with the RETURN if program execution is to return to the statement following the statement where the error occurred. All statement errors will be sent to this subroutine.

No buffer overflow errors may be trapped. End-of-record and end-of-file errors will be trapped by this statement.

More than one error declaration may be made, in a stack fashion. The last ERR label executed will be in effect until the ERR statement is removed from the stack. This will automatically happen at the termination of a block, if the ERR command is inserted within a block, or may be abnormally removed by use of the POP command.

ERR is an executable command and will have no effect until it is executed.

Question Mark: ?

The question mark will return a number specifying the type of error just encountered.

Format: ?

The question mark may be thought of as a variable whose value is the number of the error last reached. These error numbers are listed in Appendix D. The normal use for this command is at the beginning of an error subroutine to determine what type of error has occurred.

The "?" must be used in the error subroutine before any blocks ([ ], LOOP... NEXT, etc.) are executed. Otherwise, the "?" will not reflect the statement error number.

This "?" statement actually pops the last value off the function stack and returns it as a numerical value to the operand table. Errors that are trapped send the error number to the function stack to be used by the "?". Therefore, if the "?" is not used in conjunction with the ERR statement, some other value will be received.

```
PRINT?
X=?
```

External Subroutines and Functions

There are two types of subroutines in OPUS/TWO: The familiar GOSUB/RETURN and the expanded CALL/RET. The latter allows execution of a separate and distinct section of program code with unique variable and operand tables. Parameters may be passed between the main body of code and the subroutine. The subroutine may be entered as a part of the main code or may be loaded from the disc during program execution.

EXTernal

The external command declares a subroutine.

Format: EXT $expr_1 <, expr_2 >$

Expression 1 must be either a label indicating the start of the subroutine, should the subroutine reside within the program, or a disc file name, should the subroutine be loaded off the disc.

Expression 2 is an optional drive number or disc tag, if the subroutine is to be loaded off the disc.

OPUS will first scan the program for a matching label, indicating the start of the subroutine. If it does not exist, it is assumed that the label is a file name residing on disc. If no disc tag or number is specified, the current disc in use is searched and the file appended to the end of the object program. The start of the subroutine is stored in the function stack and retrieved with each call to that subroutine.

A variable is automatically set up with the variable name equivalent to that of the label or file name. Thus the latter must be a valid variable name. All references to the subroutine will be made through this variable. The variable may be given any value and may be treated as a passed parameter to the subroutine.

Normally OPUS object programs are declared as externals. However, any type of file is permissible, if the file type is specified.

```
EXT "TEST";
EXT "DX", 1;
EXT "ABS", "DISC1";
```

GLOBAL

The GLOBAL statement declares certain variables to be treated as global or common variables.

Format: GLOBAL identifier, variable list

The identifier must be a one character literal string identifying the GLOBAL list. Any ASCII character is permissible.

The variable list is any list of those variables whose values are to remain in common to any subroutine with the same GLOBAL identifier.

Normally, upon entry to a subroutine, a new variable table is set up for the subroutine and there is no connection between variables in the calling routine and the subroutine. The exception to this is those variables declared in the GLOBAL statement. Their values are automatically passed to the subroutine. This subroutine must also have a GLOBAL statement with the same identifier. The variable list may be different - the same values will be assigned to different variable names.

When a GLOBAL statement is encountered, OPUS/TWO will first search the stack for a prior GLOBAL with the same identifier. If it exists, the new variables are assigned the values of the prior global variables. If no GLOBAL does exist, the command enters the variable list into the stack to be used by a future GLOBAL command.

The GLOBAL command need not be the first command in a program or subroutine.

If the size of the variable list of the second GLOBAL does not match that of the initially declared GLOBAL, OPUS will only treat the smaller of the two lists as GLOBAL variables. The rest will be ignored.

Main program:

```
   10  GLOBAL "A",X,DOG,  FUDGE;  EXT"SR";
   20  X= 1.5; DOG= "HI"; FUDGE= X & DOG;
   30  CALL SR;
                .
                .
                .
```

Subroutine:

```
 1000  SUB "SR", SR;  GLOBAL "A", A,B,C;
                .
                .
                .
```

At entry of the surbourtine, A is 1.5, B is "HI", and C is "1.5HI".

CALL

The CALL command calls a subroutine previously declared in an EXTernal statement.

Format: CALL variable name <(list)>

The variable name must be identical to a label or file name previously declared in the EXTernal statement.

The list must be a parameter list of variables, expressions or constants enclosed within parentheses. These parameter values will be passed to the subroutine and assigned to variables declared in the subroutine.

Should a variable be passed to the subroutine, any changes made to the value of that variable in the subroutine (even if the name is different) are reflected in the main routine. These are treated in the same fashion as GLOBAL variables.

```
EXT "TEST";
CALL TEST(1,X,3*T+4);
```

@ Sign

The @ sign calls a function.

Format: @ variable name<(list)>

This command behaves identically to the CALL command. It is included here only to allow the user to differentiate between subroutines and functions. In actuality, the two commands may be interchanged. A function is determined by whether or not any operand is left in the operand table prior to exiting the subroutine. If one is, that value will be returned to the main routine to be used as needed.

Main program:

```
100  X = @TEST(2,5,10);
200  PRINT X;
          .
          .
          .
```

Subroutine:

```
1000  SUB "TEST", TEST, A,B,C;
1010  RET A+B+C;
          .
          .
          .
```

Upon execution, X will be printed:

```
17
```

SUBroutine

This command declares code to be an external subroutine.

Format: SUB < label,> variable name < variable list >

The label must be the literal string declared as the subroutine name, if the subroutine is included in the main body of code and is not to be loaded off disc in the EXT statement. If it is to be loaded during the EXT statement, the label is optional. The label must be identical to the string in the EXT statement.

The variable name must be the same variable by which the subroutine was called.

The list corresponds, element by element, to the list in the prior CALL or @ statement. The value of each parameter in the CALL (@) list is assigned to the corresponding variable in this list. If the lists are of different length, all values or variables in the longer list not corresponding to an element in the shorter list are ignored.

The SUB command must be the first statement in a subroutine.

If a REM statement is at the start of a subroutine, make sure that no semicolon follows it or else all passed parameters will be lost.

```
EXT "TEST";
CALL TEST (2,A,3*T,Z);
          .
          .
          .
SUB "TEST", TEST,A,B,C,DOG;
```

RETurn

This command terminates an external subroutine and returns control to the main routine.

Format: RET <list>

If the list exists, the value of these parameters will be returned to the main routine in the operand table to be used as needed.

```
RET
RET 3*A+B;
```

Miscellaneous Notes:

1. Subroutines may call other subroutines (limited only by memory size). However, if a subroutine is defined externally in one section of code and called in a different subroutine, the name must be either passed as a parameter or put in a GLOBAL statement. This is because a subroutine name is a variable and treated as such.

2. External subroutines are recursive.

Extended String Manipulation

SEEK

This function searches a string for a matching substring and returns the starting character position within the string of the substring.

Format: SEEK $expr_1$, $expr_2$

Expression 1 is the string which is to be searched.

Expression 2 is the substring which is to be sought for in expression 1.

If no match is found, the function will return a Ø (zero). Otherwise, it returns the first character position of the substring within the string.

```
X="HORSE";
PRINT SEEK X, "SE";
4
```

Extended File and Disc Manipulation

Dimensioned Files

There are now two types of data files in OPUS/TWO: The standard random file of type F and the new dimensioned file of type D. Unless the D type is specifically declared, the F type will be assumed as a default. Refer to Section VIII. for a description of the F type file.

The dimensioned file is treated much as a matrix. There exist from one to six different dimensioned fields, with each field having the range from Ø to 9. The logical record number defines the element of this psuedo-matrix, where each digit in the number corresponds to one dimension.

For example, a matrix element M(1,2,3) can be compared to a logical record number of 123.

The purpose of dimensioned files is to allow the user to directly access logical records where the logical record is an identifying number with each digit having a separate meaning. A good example of an application is a general ledger system, where the account number may have digit 1 representing the account type (equity, assets, income and expense), digit 2 specifying the sub-account number, and digit 3 the contra-account status. Instead of either going through a long hashing routine or opening an F-file to handle the maximum account number, a D-file allows the user to treat the account number as a logical record number. There is no room allocated on the disc for 'gaps' in the logical record sequence.

The following procedure should be followed in handling D-files.

1. Opening a D-file

```
OPEN "TEST\D",L,D
```

The file type D must be specified after the name of the file. L is the number of logical records/sector (like the F-files). D defines the dimensions of the file. Each digit in the number declares the maximum element for the corresponding dimension.

Example:

```
OPEN "TEST\D",1,234

There are three dimensions:

Dimension 1 has elements from 0-2
Dimension 2 has elements from 0-3
Dimension 3 has elements from 0-4
```

If the file were listed sequentially, the record numbers would go:

```
0,1,2,3,4,10,11,12,13,14,20,21,22,23,24,
30,31,32,33,34,100,101,102,...,234
```

2. Assigning D-files

```
ASSIGN "TEST\D",F,D
```

The D type must be specified in the assignment. F is the file position number. D is the drive number or disc tag.

3. Accessing D-files

D-files are accessed the same way as F-files, by giving the logical record number.

An End-Of-File will occur if a record number is specified that is not within the range of the dimensions.

Remember that logical record Ø is a valid record number.

For reading sequentially through a D-file, it is urged that the user utilize the REC command for finding the next logical record number.

Expansion of the Disc Command

In OPUS/ONE, the command DISC will always read tables from the specified disc drive into memory. The drive number is mandatory. It is possible to change diskettes simply by inserting a new diskette and using this command.

In OPUS/TWO, the command DISC will only read the tables on the disc if the disc has not been previously enabled. Otherwise, it is assumed that the table is already residing in memory. The SWAP command must be used to change diskettes under OPUS/TWO. In addition, the parameter to the DISC command may be either a drive number or a disc tag.

SEQuential

This function returns the next logical record number of a file that has had data written in it.

Format: SEQ $expr_1$, $expr_2$

Expression 1 is the file number of a file previously assigned.

Expression 2 is the starting logical record number from which OPUS is to begin its search for a record with data.

The number returned by the function will be a logical record number that contains data.

If an invalid logical record number is given or the function fails to find a record with data, an End-Of-File (EOF) will be reached. In such a case, no value will be returned to the operand table.

```
PRINT SEQ 1,1;
X=(SEQ 3,R);
```

ESEQuential

This function returns the next logical record of a file which contains no data.

Format: ESEQ $expr_1$, $expr_2$

Expression 1 is the file number of a file previously assigned.

Expression 2 is the starting logical record from which OPUS is to begin its search for an empty data record.

The ESEQ function will return the logical record number of the next record with no data.

If an invalid logical record number is given, or the function fails to find an empty record, an End-Of-File (EOF) will result. In such a case, no value will be returned to the operand table.

```
PRINT ESEQ Z,5;
X=(ESEQ 1,X);
```

RECord

This function will return the next sequential logical record of a file.

Format: REC $expr_1$, $expr_2$

Expression 1 is the file number of a file previously assigned.

Expression 2 must be a valid logical record number.

This function will return the next sequential logical record number from the given record number.

Obviously, for type 'F' files, the record numbers are in sequential order and the REC command will simply increment the record number by one. However, for type 'D' files, where each digit in the record number has its own upper boundary, the REC command is useful in retrieving the next logical record number.

```
ASSIGN "FFILE",1; ASSIGN "DFILE\D",2;
PRINT REC 1,5;
   6
PRINT REC 2,156;
   200   (DFILE OPENED TO 256 MAX LR)
```

TAG

This function returns either the tag of a specified disc or the disc number of a specified tag.

Format: TAG $expr_1$, $expr_2$

Expression 1 must have an integer value of one (1), two (2) or three (3):

If expression 1 is one (1), expression 2 must be a disc tag of 1 to 7 ASCII characters previously tagged to a diskette. The function will return the drive number (Ø to n) of the drive where the tagged diskette is located. If no diskette with this tag is present, a -1 will be returned.

If expression 1 has a value of two (2), expression 2 must be a disc drive number. The function will return the tag on the diskette currently in the specified drive as a string. If the drive is not enabled, a -1 will be returned.

If expression 1 is three (3), expression 2 must be a valid disc tag. The diskette in the drive last declared in the DISC command will be given this disc tag. No value will be returned to the operand table.

SWAP

The SWAP command must be used to swap diskettes in and out of drives.

Format: SWAP expr

The expression must be a drive number or disc tag, identifying the disc that is to be swapped (current diskette removed and a new one inserted).

```
SWAP Ø;
SWAP"XXX";
```

A new disc need not be inserted. It will not be enabled by this command.

The SWAP command utilizes the same Disc Swap Routine that is automatically executed during disc commands that request a disc tag not currently on the system. OPUS will type:

SWAP DISC & HIT RETURN

The user should swap the diskette and press the carriage return.

Multi-User OPUS/THREE Only:

The command is a function that will return a 1 (one) if the disc may be swapped and a Ø (zero) if the disc is being used by another job. In the latter case, OPUS will print:

DISC IN USE BY JOB XX

DO NOT swap the disc.

NOTE: IT IS EXTREMELY IMPORTANT THAT THE DISC BE SWAPPED ONLY UNDER THESE CONDITIONS, BECAUSE TABLES MUST BE UPDATED ON THE DISC.

Machine Code Subroutines

The user has the capability of calling machine code routines from an OPUS/TWO program. There is one command with which to call this subroutine.

Machine CALL

Transfers program execution to the user machine code subroutine (executes a machine code CALL).

Format: MCALL $expr_1$<($expr_2$, $expr_3$)>

Expression 1 must be the absolute machine code subroutine starting memory address. The subroutine must lie at an absolute location in memory.

Expression 2 has a value that will be converted to a two-byte number and stored in the B and C Registers. These registers will be passed to the subroutine. String values will first be converted to an OPUS number.

Expression 3 has a value that will be converted to a two-byte number and stored in the D and E Registers. These registers will be passed to the subroutine. String values will first be converted to an OPUS number.

OPUS will execute a machine code CALL (315 in Octal) to the subroutine. The subroutine must terminate with a RETurn (311 in Octal).

Overlays

In OPUS/TWO, an overlay is defined as a buffer of memory allocated at the end of the object program to be utilized for program or data storage.

OverLAY

Sets up a memory buffer of a specified number of bytes.

Format: OLAY expr

The expression must be the number of bytes determining the size of the buffer.

The buffer will be located at the end of the current OPUS object code. The buffer may be used for any purpose, but its primary function is as an area into which OPUS object code subroutines may be loaded. This is useful if different code needs to be loaded at different times during program execution. The normal loading procedure will append the new code to the end of the object code program, never clearing out unused code, whereas code loaded into the overlay will always replace any previous code located there.

To load a file into the overlay, the user must use either the LOAD or EXT commands, followed by a backslash, the file type and "O". The "O" specifies the latest overlay block.

```
OLAY 2000;
EXT "TEST\OO";
```

More than one overlay may be declared, but they are treated in stack fashion. That is, the last overlay declared will be used until the overlay is cleared (by use of the POP command or, if the overlay was declared within a block, the exit from the block will clear the area).

An error will occur if the length of the file to be loaded into the block exceeds the size of the buffer.

An OPUS program loaded into an overlay will be followed by an end of program marker. Thus if more code is loaded normally following the overlay, no command in previous code referring to a label in the latter code will be successful. All label references scan the program for a matching label until and end of program marker is reached.

The start of the last overlay declared may be referenced with a label of "?". For example, the command, GOTO "?", will automatically send execution to the start of this overlay even though there is not a matching label of "?" at the start of the overlay.

Miscellaneous Statements

DATA

This command will store data directly in a matrix.

Format: DATA variable : list

The variable must be a matrix variable previously defined in a dimension statement.

List is any number of expressions, variables, or constants that are to be stored in the matrix.

Each computed value in the list will be stored in the operand table and then all values in the operand table dumped into the matrix. Should the number of values exceed the size of the matrix, only as many as needed will be withdrawn from the operand table. Should the number of values be less than the size of the matrix, the remaining matrix elements will remain unchanged.

Because all values are stored in the operand table, it is easy to overflow the table if there are many values. Use the SET command to change the size of the table if necessary.

Given a matrix, M(D1,D2,...,D_n), if K is the dimension position, then D_K is the maximum number of elements in the dimension. The data statement will fill the matrix in the following order:

```
M(1,1,...,1,1)
M(1,1,...,1,2)
    .
M(1,1,...,1,Dn)
    .
M(1,1,...,2,1)
    .
M(1,1,...,2,Dn)
    .
    .
    .
M(D1,D2,...,Dn)
```

Example:

```
DIM M(2,2);
DATA M: "BOO",X*Y+4,5.666,"X"&Z;

M(1,1)  IS  "BOO"
M(1,2)  IS  X*Y+4
M(2,1)  IS  5.666
M(2,2)  IS  "X"&Z
```

POP

This command will cause program execution to abnormally leave a block.

Format: POP expr$_1$ <, expr$_2$>

Expression 1 determines the block level that is to be exited. It may be one of the following:

1. The number of blocks to exit -- an integer greater than or equal to 1.

2. The block type to exit -- a one character string defined below.

Expression 2 is an optional label defining the location to which program execution is to transfer, once it has left the block. If no label is given, execution transfers automatically to the statement following the block terminator.

The POP command is similar to a GOTO in that it unconditionally transfers program control. However, using a GOTO to leave blocks will usually result in unfortunate errors because the stack still contains block terminators and other data that will be picked up by later statements. The POP command clears the stack and leaves everything in normal condition. Examples:

```
POP 2          Exits 2 blocks
POP 1,"LABEL"  Exits 1 block and sends control to "LABEL"
POP "L"        Exits a LOOP/NEXT block
```

Table of Block Types for the POP Command

Type	Initiator	Result of POP Execution
B	[	Execution leaves the bracket block
G	GOSUB	Execution leaves the subroutine
L	LOOP	Execution leaves the LOOP...NEXT block
O	OLAY	Removes last overlay buffer
R	CALL, @	Execution leaves the subroutine
S	:	Returns original start of operand table
W	WHILE	Execution leaves the WHILE...CONTinue block

Execution of the POP command will start through the function stack and remove parameters as requested. This stack has additional functions other than those listed above, but, although they will be removed, they do not count as true block POPs.

If a certain number of blocks, n, is specified, OPUS will remove the first n blocks among those listed above on the stack, starting with the current block. If n is greater than the actual number of blocks on the stack, the stack will be completely cleared, but no error will occur.

If the block type is specified, OPUS will continue to POP all blocks off the stack until it encounters the requested type. A "STACK OVERFLOW" error will occur if the type is not present.

Byte IN

This command may be used to read in one or more distinct bytes from an input device.

Format: BIN <expr:> variable list

The expression is the device number from which the byte is to be received. If it is not given, the current input device is assumed. The variable list contains all variables which are to be given the value of the byte as it is received. The first variable will receive the first byte, etc. The value of the byte will be the ASCII decimal number representation from Ø through 255 (parity bit included). The byte will not be echoed on the output device. OPUS will look only for as many bytes as there are variables. A matrix variable may be specified. See Appendix E. for an ASCII table. Examples:

```
BIN X;
  (user types "A")
PRINT X;
65

BIN 3: A,B,C;
  (received "1", "2", and "3" from Device 3)
PRINT A,B,C;
49   50   51
```

ASCII Program Files

OPUS programs written in ASCII code (under the A.S.I. Text Editor) may be loaded by OPUS/THREE and compiled into normal OPUS source code. ASCII files have the type identifier of "$". This procedure would be as follows:

LOAD "TEST\$"	Loads the ASCII file into memory.
COM "A"	Compiles the ASCII to source and automatically assigns line numbers starting with 10 and incrementing by 10.

The ASCII program buffer may be cleared by:

```
NEW "A"
```

If compiled, the source and/or object will remain.

OPUS/THREE has no capability of creating ASCII program files, but only of reading them in and converting them to source.

An ASCII file contains each character in the program in its ASCII code representation. The ASCII file must not contain any OPUS line numbers.

TRACE

This command allows a program to be traced, step by step, through execution.

Format: TRACE expr

The expression must have either a TRUE (non-zero) or FALSE (zero) value.

TRUE : TRACE function turned on
FALSE: TRACE function turned off

Upon enabling the TRACE function, with each statement executed, OPUS will print the location in the program of the statement and the statement name, and will go into an input mode. In standalone OPUS, all locations will be printed as a two-byte decimal address; in re-entrant OPUS, they will be either Octal or Hex, depending upon the mode.

In the input mode, the user has four options:

1. Carriage Return: OPUS will go to the next statement.

2. O: The contents of the operand stack will be printed. The first value printed is the first on the stack. The format is:

   location<buffer>value

   The location gives the memory address of the start of the value. The buffer is a character designating where the value is stored. It may be one of the following:

   P Program area
   V Variable Value Table
   C Constant Table

   The value is the value of the operand that is to be used by the current statement. Remember that code is stored in Postfix notation and operands precede the operator.

3. F: OPUS prints the number of bytes free in the job area. This is the number of bytes between the function stack, which works from top memory downwards, and the Variable Value Table, which is the last buffer following the program, working upwards to the function stack.

4. S: OPUS prints the contents of the function stack. The first value printed will be the last item entered on the stack. The format is:

   type value<buffer>

   The type is a one-character code defining the operation that used the stack at this point. The value depends upon the type and will be printed as a two-byte number as it is stored in the function stack. The high order byte will be followed by the low order byte. In standalone OPUS, the numbers will be in decimal; in re-entrant OPUS, the numbers will be Octal or Hex, depending upon the mode.

The following table lists possible types and values if the "S" option is used:

Type	Operation	Value
B	[...] Block	Initial start of Operand Table
G	GOSUB...RETURN	Return address
L	LOOP...NEXT	Location following LOOP
O	OLAY	Number of bytes in overlay and starting location
R	CALL/@...RETurn	Return address
S	:	Initial start of Operand Table
W	WHILE...CONTinue	Starting location of block
A	Block Operations	Initial address in Variable Value Table
C	GLOBAL	Identifying label character
E	ERRor	Location of error routine
I	IF...ELSE	TRUE (1) or FALSE (Ø) condition
X	EXTernal	Location of external subroutine

The buffer gives the general location that is affected by the value. The possibilities are:

P	Program area (or non-applicable)
O	Operand stack
C	Constant Table
N	Variable Name Table
V	Variable Value Table

After any of the three latter options are requested, OPUS will return to the input mode for further options. When through with that statement, the user must hit carriage return in order to continue. Example:

```
LIST
10  TRACE 1;
20  LOOP I, 1TO 2;
30  PRINT I* I;
40  NEXT ;
50  TRACE Ø;
60  END ;

FINE
COM

FINE
RUN
Ø71/361  ;
Ø71/366  LOOP
0                  Ø76/127 < V >          Ø
                   Ø71/364 < P >          1
```

```
S
F                    13478
Ø71/372  TO
S                    A   Ø76/126  <V>
                     L   Ø71/367  <P>

Ø71/373  ;
Ø71/376  *
O                    Ø76/127 <V>          1
                     Ø76/127 <V>          1

Ø71/377  PRINT
1
Ø72/ØØØ  ;
Ø72/ØØ1  NEXT
Ø71/372  TO
S                    A   Ø76/126  <V>
                     L   Ø71/367  <P>

Ø71/373  ;
Ø71/376  *
Ø71/377  PRINT
O                    Ø72/Ø7Ø <C>          4

4
Ø72/ØØØ  ;
Ø72/ØØ1  NEXT
Ø71/372  TO
O                    Ø71/37Ø <P>          2

Ø72/ØØ2  ;
F                    13478
Ø72/ØØ6  TRACE
O                    Ø72/ØØ4 <P>          Ø


FINE
```

OPUS will trace all statements either in the program or in command mode until the user either gives a "TRACE Ø" to disable it or types "NEW". TRACE is automatically turned off with the NEW command.

This TRACE function can be a valuable tool for both debugging and finding out how OPUS executes statements. The user is urged to utilize it freely.

Multi-User Commands

The following two commands are specifically implemented for use in OPUS/THREE under the TEMPOS Multi-User/Multi-Tasking Operating System.

TIME

This function will return the current time in seconds, minutes, or hours.

Format: TIME expr

The expression must be an integer from 1 to 3 that designates which of the following times will be returned to the Operand Table:

1 Second
2 Minute
3 Hour

The designated value will be returned to the Operand Table for use by another operation. The value is dependent upon the time that was entered when TEMPOS was first brought up.

HANG

This command should be used to lock out data files from other users during critical READ/WRITE operations.

Format: HANG $expr_1$, $expr_2$

Expression 1 is a file position number designating the data file that is to be locked out. The file must have been previously assigned to this number.

Expression 2 must be either a true (non-zero) or false (zero) value, where:

True — The operation will attempt to lock out the file. If no other user has previously locked out the file, other users attempting to HANG the file will be put in a wait mode, until a HANG operation with a false value is executed.

If another user has previously locked out the file, this operation will be suspended until the other user releases the file with a false value.

False — This operation will release the file. Any other user currently locked out will be released and allowed to access the file. If more than one user is waiting for the file, the first user to be released is determined by the sequence of the clock - who gets the next time slice.

The HANG operation does not prevent any user from reading from or writing to the file. It simply locks out an other user who is also attempting to use the HANG operation.

The HANG command should be used in any program that may be run simultaneously by different users and that changes data in a file. This will prevent situations such as the following:

User 1 Reads a logical record and makes changes in it.
User 2 Reads the same logical record and makes different changes.
User 2 Finishes first and writes the record back on the file.
User 1 Finishes and writes the record back.
Result: The changes by user 2 have been completely wiped out - do not exist any more.

If there was a HANG operation prior to accessing the file, user 2 could not have even read the record before user 1 had finished making changes and updating the record.

It is quite important that a HANG operation with a false value be executed after the critical operation. If this is not done, the file will be permanently locked out until the user terminates the job.

Note that it is possible to lock oneself out by executing two HANG operations with a true value sequentially.

The HANG command will clear all flags set by the EFILE statement. EFILE must again be declared following a HANG release.

Machine Code Relocatable Files

Machine code relocatable files may be loaded, executed, and dumped from OPUS/THREE. The relocatable file must have been created with the A.S.I. Assembler. After being located at an absolute address, these files are in directly-executable machine code.

The relocatable file must be loaded and treated as an external subroutine by using the EXTernal command with a type designation of "R":

```
EXT "TEST\R";
```

The file will be appended to the end of the OPUS object program and all memory address references updated to reflect the location.

Relocatable files may be loaded into overlays:

```
EXT "TEST\RO";
```

The overlay buffer must be large enough to hold both the code and the relocatable table of the file.

The name of the subroutine must be the name of the relocatable file and a valid variable name.

To execute the machine code subroutine, the MCALL command must be executed, using the name of the subroutine:

```
MCALL TEST;
MCALL TEST(A);
MCALL TEST(3,50);
```

OPUS will send control to the subroutine. The values within the parentheses are optional. The first value will be assigned to the B and C register pair, the second assigned to the D and E register pair. Only integer values from Ø through 65535 will be recognized. The machine code subroutine may do what it likes with these values.

To return control to OPUS, the subroutine must terminate with the RETurn (311 in Octal) instruction. All register values may be destroyed in the subroutine. The subroutine may call external subroutines either in the primary operating system or in OPUS.

A relocatable file contains two sections:

1. Actual machine code
2. Address relocatable table

The second section contains all locations in the code to be updated prior to execution.

With a machine code program, it is possible to separate a relocatable file into two files reflecting the above. Under OPUS, these two files may be re-combined to form the original relocatable file. The machine code portion must be a file of type "S", OPUS source, and the relocatable portion a file of type "O", OPUS object. They may be combined into a single type "R" file by the following:

```
GET "CODE";        (machine code)
LOAD "RELOC";      (relocatable table)
DUMP "FILE\R";     (creates relocatable file "FILE")
```

The need for this may never arise, but may be useful for transferring relocatable files from one disc to another using only OPUS.

OPUS Subroutines

The following subroutines in OPUS may be called by any user machine code program. OPUS must be in memory. The user must simply use a CALL command followed by the name of the subroutine. Upon loading the program, the address of the subroutine will be inserted after the CALL. The user should assume that all register values may be lost during the subroutine.

The specifications for the subroutines, as shown in the following table, are:

Label	Name of the OPUS subroutine
Description	Description of what the subroutine accomplishes
Required	These registers must contain these values when the subroutine is called
Returned	The subroutine will return these register values from the subroutine

The following abbreviations are used:

'R'-	Register pair 'R' points to memory location
'R'=	Register (pair) 'R' has the specified value
BUF	Miscellaneous buffer within the program used temporarily by the subroutine
VALUE	The start of an OPUS number or string constant
#	An OPUS number (the format is described in Section II.)
$	An OPUS string (the format is described in Section II.)
TYPE	Number identifier (number of bytes in mantissa) or string identifier (number of bytes in string + 128)
START	The starting location of the buffer containing data
END	The last location of data in a buffer plus one
Z:	Zero status bit high
C:	Carry status bit high
NC:	Carry status bit low

Label	Description	Required	Returned
OPSST	Converts OPUS constant to string, constant either $ or #	DE-BUF HL-VALUE+1 A=TYPE	HL-#
OPSNO	Converts OPUS constant to number, constant either $ or #	DE-BUF HL-VALUE+1 A=TYPE	HL-$
OPSSTR	Converts OPUS number to ASCII characters	HL-# DE-BUF	HL-START DE-END
OPSHL	Finds sign of OPUS number	HL-#	Z: =Ø C: <Ø NC: >Ø
OPSCONA	Converts byte in Register A to an OPUS number	A=?	HL-#
OPSOSN	Retrieves one value from operand stack in number format	None	HL-#
OPSOSS	Retrieves one value from operand stack in string format	None	HL-$
OPSRET	Returns value to operand stack, either number or string	HL-VALUE	None
OPSCMP	Compares two values, either number or string	HL-VALUE DE-VALUE	Z: HL=DE C: HL>DE NC:HL<DE
OPADDR	Location of OPUS Statement Table -- the starting address of each statement subroutine in OPUS resides in this table. Two bytes are required for each address. The location of the statement in the table is the corresponding statement number listed in Appendix D. This is not a subroutine to be called, merely a location to be accessed to find statement routines. To call statement subroutines, values must be pushed onto the operand stack to meet the requirements of the particular statement.		

XV. THE WHYS AND WHY NOTS OF COMMON PROBLEMS

The following section is designed to give the user more detailed insight into problems or idiosyncracies that may often have one tearing one's hair out. If the user is having trouble with a program, this section should be the first place to look. The following topics are covered:

A. How imbedded spaces within key statements cause strange and unusual program execution

B. Why jumping out of or into the middle of blocks is not always a good idea

C. How to get an error with a valid command in command mode

D. The difference between BASIC and OPUS program SAVE and KILL commands

E. How to destroy a data diskette

F. How to get OPUS to ignore every command

A. IMBEDDED SPACES WITHIN KEY STATEMENTS

Have you ever tried the following program:

```
10  "START";
20  PRINT "THIS IS A TEST";
30  GO TO "START";
```

and, upon execution, had this happen:

```
THIS IS A TEST
STACK OVERFLOW!
```

Obviously the program should indefinitely print out "THIS IS A TEST", right? Upon careful scrutiny of line 30, one notices that there is a space between the GO and the TO of the GOTO statement. This is an error and will cause problems. Spaces within key statements will always cause problems unless the user is consciously aware of the act.

What is happening in the above program? Because of the space, OPUS assumes that the GO is a variable name and of course the TO is a valid statement by itself (LOOP I,1 TO n). But a TO statement requires a prior LOOP statement for execution, and, if the LOOP is not present,will bomb on a "STACK OVERFLOW" error. (The LOOP statement pushes the variable on to the stack and the TO statement looks for that variable).

Unfortunately, a listing of the above program will not show the imbedded space:

```
LIST
10  "START";
20  PRINT "THIS IS A TEST";
30  GOTO "START";
```

Note the following rules that OPUS follows when listing a program:

1. A space will always be typed after every statement key-word. Note that this includes semicolons, commas, and parentheses.

2. No spaces will be typed after variables or constants.

Thus in the above program, no space is typed after GO - it is a variable - but a space is typed after TO since it is a statement word. The crucial point is upon entering the line in the program. Upon a line entry, the line is scanned for all commands, constants, and variables and stored in a compacted source form.

Although this will often cause problems, the user can look at it from a different point of view and see it as a way to use reserved statement words within a variable name. By typing a space in the middle of a statement, thus breaking that statement into two non-statement names, it is possible to have a variable "GOTO", "PRINT", "XLOOP", etc. It is important that the statement word be broken in a place that does not produce a new statement, as the program above shows. If the user had typed in "GOT O", then indeed the variable "GOTO" would have been created. Upon listing

this version, line 30 would have been:

```
30  GOTO"START";  (note the lack of a space after  GOTO )
```

Further examples:

```
PRINT   statement "PRINT"
PR INT  variable "PR", statement "IN", variable "T"
PRI NT  variable "PRINT"

XLOOP   variable "X", statement "LOOP"
XL OOP  variable "XLOOP"
XL,OOP  variable "XL", variable "OOP"
```

B. ILLEGAL BLOCK ENTRIES AND EXITS

One of the most common forms of a statement used by beginning OPUS programmers is the following:

```
100  IF X#Y [ GOTO "END" ];
```

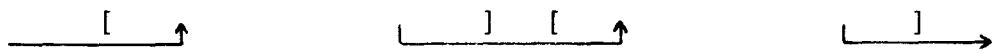
What the user had in mind is obvious - If X does not equal Y, transfer program execution to the label "END". What could be simpler? Nothing, except that problems may arise. The program is jumping out of the middle of a block (in this case a bracket [ ] block) without successfully terminating that block. The stack is used by OPUS to store the start of the block (with any required parameters). If the end of the block is not reached (here it would be:]), this parameter on the stack will not be removed and OPUS assumes that the program is still in the block at execution of the code following the label.

If this only happens once or a couple of times, the user will probably not be aware of anything wrong. Who cares if the program is nested a couple of blocks deep. However, continuous use of this statement will quickly eat up memory and a "STACK OVERFLOW!" error may occur.

Before commenting on possible solutions, it is necessary to take a closer look at the structure of blocks. A block may be not only a bracket block but may be a LOOP... NEXT, WHILE...CONTinue, or subroutine block.

A block is a unique section of program code that has a specific reason for being treated in this manner. The best program code will always enter and leave a block by first executing the block initiator ([, LOOP, WHILE, GOSUB) and later executing the block terminator (], NEXT, CONT, RETURN). Attempts to enter the middle of a block or leave the middle of a block will almost always result in errors unless the jump is made to the same type of block at the same block level.

The following illustration may clarify this concept:



1. Illegal 2. Legal 3. Illegal

1. The program jumps from code not in a block into the middle of a block. When the block terminator is reached, a "STACK OVERFLOW" error will probably occur because there is no matching block initiator.

2. The program jumps from the middle of one block into the middle of another block of the same type. No error will occur because the program is at the same block level. The first block terminator and the second block initiator are essentially not there.

3. The program jumps out of the middle of a block into code not included within a block. No error will occur immediately. However, since the block was never terminated, it is still reserving room in the stack. Continuous use of this statement may cause a "STACK OVERFLOW" error. Also, if the first block was nested within another block, a "STACK OVERFLOW" error may occur when the terminator

for this latter block is reached. This terminator will look for the matching initiator on the stack. If this initiator is not the first item on the stack, an error may occur. (In this case, the first item would be the initiator of the first block that was illegally exited).

Back to the problem above with the GOTO. What are the possible solutions? One quick and dirty solution to this problem is to put brackets around the label:

```
100  IF X#Y [ GOTO "END" ];
 :
 :
 :
999  [ "END" ]; . . .
```

As should be apparent, the program will go to the "END" label and execute the right hand bracket ( ]), thus terminating the block that was entered following the IF statement ([ ). This works quite well if the program always jumps to this label from the middle of a bracket block. If any attempt is made to jump there from a GOTO that is not in a bracket block, a "STACK OVERFLOW" error will occur when the "]" is executed and there was no previous "[".

The best solution to the problem of jumping out of or into blocks is to re-write the program code so that the need does not arise. Take a long look at the program and see if it would not be possible to take advantage of the "IF . . . [ ]ELSE [ ]" or "WHILE . . . CONT" capabilities. It is the writer's experience that any program can be written in OPUS without a single GOTO statement. See the sample programs in the Appendices for ideas on writing GOTO-less programs.

C. CLEARING THE OPERAND TABLE

Anyone who has programmed in OPUS is well aware of the situation where a valid command has been entered in command mode (such as LIST or NEW) and a statement error is returned. Repeating the command normally works. What is happening?

Look at the operation preceding the command just entered. Was there an error in that statement, or did that operation leave a value in the operand table (such as "=")? More than likely so. The secret lies in what was left in the operand table. Remember that OPUS executes code in a Postfix sequence, i.e., all operands that are to be used by a statement are first stored in the operand table prior to executing the statement. When a statement is executed, it will pull from this operand table all parameters it may need. If optional parameters are allowed, and there is anything in the operand table, the statement will pick them up as user-desired parameters.

When an error occurs in a statement, some or all of the parameters used by that statement may be left in the operand table. Upon executing another command, this command will retrieve those left-over parameters and attempt to use them as its own parameters. More than likely, this will cause another error, since these parameters will probably not meet the format required by the current command. One may ask why errors do not clear the operand table automatically. They easily could. However, many times it is useful in debugging a program to know what was in the operand table at the time of an error (a PRINT statement works quite well in printing out these values).

Similarly, a parameter will be left in the operand table by any operation that is specifically designed to return a value, such as assignment or any binary or unary operation. The following operation will pick up this parameter as its own, possibly resulting in an error.

What is the solution? As mentioned above, a repeat of the command that produced the error will work. However, it saves time and frustration simply to enter a semicolon (;) prior to the command. Remember that the semicolon is a valid operation (not only a delimiter), functioning specifically to clear the operand table of unnecessary parameters.

Keep in mind that operating in command mode is almost identical to running a program. In programs, the user normally enters a semicolon after every operation: thus, in command mode, semicolons are likewise needed to clear miscellaneous parameters from the operand table.

Examples of operations in command mode ("FINE" not printed here):

```
A="HORSE"      (leaves  HORSE  in the operand table)
PRINT "DOG"
HORSE   DOG    (prints all values in the operand table)

X+Y            (leaves sum in the table)
NEW
NEW ERROR      (new attempts to use sum of X and Y as a parameter)
```

```
READ 1,1:X,Y
READ ERROR    (no file assigned to file number 1)

;NEW          (successful - if no semicolon, a new error would have occurred
               since the read statement above left the list parameters in the
               operand table)
```

D. THE FORMAT OF DISC STATEMENTS

Many BASIC programmers often have a problem with OPUS when trying to SAVE, KILL or retrieve a program from the disc. In most basic languages, the format of a SAVE command is similar to:

```
SAVE-TEST          ( TEST  is the name of the program)
```

In OPUS, the format is:

```
SAVE expression    (the value of the expression is  TEST )
```

The problem seems to arise when the programmer types:

```
SAVE TEST          (example 1)
```

And no program by the name of TEST appears in the library of the disc, although a zero most likely will.

This seems logical if one notes that it is the value of the expression, not the expression itself, that becomes the name of the program. In example 1, TEST is a variable because it is not enclosed in quotes. The value of that variable will probably be zero, unless the user had previously given it another value.

Here are various examples of saving a program called TEST:

```
SAVE "TEST"        (value is the literal string "TEST")

X="TEST";
SAVE X

X="TE"; Y="ST";
SAVE X&Y
```

This also applies to the CSAVE, GET, LOAD, KILL and OPEN commands.

E. THE CARE AND FEEDING OF DATA DISKETTES

One day the programmer sits down at the terminal all ready to create more new and wonderful programs. Upon accessing files and/or programs on the current development disc, things go hay-wire, such as:

1. Upon listing a program that was on the disc, lots of garbage spews forth.

2. A particular logical record in a file may suddenly contain something other than what was previously written into it.

3. A LIB of the disc reveals strange programs, most often lots of zeros.

4. A disc failure occurs, listing the track as some outrageous number such as 200 or 300.

Panic sets in, followed by frustration, ending with anger as the realization hits that a lot of previous work has gone for naught.

The problem? Carelessness (or ignorance) in swapping diskettes in and out, not closing data files that may have been open before halting the system, or a hardware failure causing the system to crash at a critical point.

This may not help the programmer solve the problems with the destroyed disc, but carefully reading the following will help prevent similar problems in the future.

Each data diskette contains a map of all sectors on the disc which are currently in use by a program or file. This resides on track Ø on the first two or three sectors. Upon enabling a disc with the DISC command this map is transferred into memory. Any operation which saves a program or writes data into a file logical record uses this map in memory (!) to determine available sectors in which to put the data. The KILL and PURGE commands use this map to return sectors to an empty status.

Problems will occur when:

1. The system is crashed before the current map in memory is written back on the disc. The old map will be used the next time the diskette is accessed, thus inaccurately reflecting the status of empty and full sectors.

2. The diskette is removed and another inserted without going through the correct disc swapping procedure. The map of the first diskette is written onto the second diskette. New programs or data written onto this latter diskette may well write over programs or files already there, thus causing chaos on the disc. Different files may have logical records in the same sector. File data could be located within a program, causing a disc failure the next time the program is retrieved. A program may suddenly contain part of another program or garbage if the bytes are not lined up correctly.

The updated map is written back on the diskette during various operations:

1. Immediately following a SAVE, CSAVE, DUMP, KILL or OPEN command.

2. Upon execution of the CLOSE command. The map will be written back on each disc determined by the file(s) specified or all files if no parameter list is given.

3. Upon executing one of the following commands if a file had been previously assigned:

   NEW, COM, GET, LOAD, DEL, BYE

4. Upon adding, changing, or deleting a line in the source program if a file had been previously assigned.

With the understanding of when the map gets updated on the disc, by studying the following, and assuming no hardware failures, data diskettes will always be good:

PREVENTIVE MEASURES - OPUS/ONE

1. Upon inserting a new data disc into a drive, before doing anything else, type the command DISC with the specified drive number or tag. This will enable the diskette by bringing the map into memory.

2. Before removing a data diskette or crashing the system, be sure to CLOSE all files. Even if none have been open, it cannot hurt to get in practice.

3. Once a disc has been enabled with the DISC command, do not use this command again with the same diskette unless all files have been closed with the CLOSE command. This command will always read the map from the disc and put it in memory.

PREVENTIVE MEASURES - OPUS/TWO

1. Before removing or inserting a data diskette, always give the command SWAP with the specified drive number. All diskettes will be updated that had files assigned on them. The system will print:

   SWAP DISC & HIT RETURN

   Remove the current diskette (if any) and insert the new diskette. Hit carriage return to the prompt. The system now knows there is a new diskette in the drive. The next attempt to access this disc will automatically first enable it by bringing the map into memory.

2. Before crashing the system, CLOSE all files to be sure that all diskettes are updated.

3. The DISC command in OPUS/TWO differs slightly from that in OPUS/ONE. It will only retrieve the map from the disc and put it in memory if the diskette has not been previously enabled. If the diskette had been enabled, it will simply declare

the specified drive to be the default diskette (to be accessed when no other disc is specifically stated). Therefore note that if the user has enabled two drives and inserts a different diskette in the one not currently in use, failing to use the SWAP command, typing DISC D for this disc will not (!) enable the new diskette. The map of the previous diskette in this drive is still assumed to be the current map.

TIPS IN GENERAL

1. It is recommended that disc tags instead of drive numbers be used in specifying discs as much as possible. If a diskette suddenly does not respond to a tag, the user may be sure that the map from a different diskette has been written on this one. The tag of a diskette is included within the map, to be read into memory when the diskette is enabled.

2. Once it becomes apparent that a diskette has not been properly updated, do not write on this disc again. Attempts to do so will simply destroy it further. Retrieve as many programs and as much file data as possible off the diskette and save it on a new one. The diskette must be re-formatted before it can be of much use again.

F. WHY OPUS WON'T DO ANYTHING

Several times a user will plunge through the System Generation Routine, generate the tailored version of OPUS, bring OPUS up, and OPUS will respond to no command other than to say "FINE".

The top page of memory has been entered incorrectly during system generation, The above will happen if more memory is specified than is actually in the system. OPUS uses top memory for the stack and consequently for compiling all commands. If the memory does not exist, the commands are not compiled and thus not executed.

The solution is simply to go through the System Generation Routine once more, paying close attention to the memory specifications, and entering the correct number.

XVI. GLOSSARY

XVI. GLOSSARY

,

The comma is used to delimit lists of variables, constants, or operations.

Format: $expr_1$, $expr_2$, $expr_3$, . . . $expr_n$

Where n = the number of expressions in the list.

Generally, the comma will be used simply to make source code easier to read. When a program is compiled into object code, all commas are deleted. Therefore, the comma is not an operation and has no function except as a separator. At no time is it required by OPUS.

```
PRINT A, "HORSE",CAT
10   HORSE    555
```

:

The colon is used to delimit list fields for a particular operation. Several commands require the following:

Format: command $list_1$: $list_2$

$List_1$ contains one or two expression values designating such things as device number, logical record, or label. $List_2$ contains other operands required by the command for successful operation.

Append

This term refers to appending, or adding, one program to the end of another. The GET and LOAD statements will both append a program on a peripheral device to a program currently in memory. There is no separate command for append.

Argument

The argument of an operation may be defined as the value of the expression required by an operator for execution. Thus the argument of the operation "LEN A" is the variable "A".

Array

An array is defined as a one dimensional matrix.

ASCII Code Representation

Every character in OPUS is stored internally as a number from Ø to 255. This number is called the ASCII (American Standard Code for Information Interchange) code representation of the character. Relational operators comparing string values use this numeric code to determine the relationship of one string to another.

Assignment

A variable or matrix element may be assigned values by using the "=" operator.

Binary Operator

A binary operator is a statement that pulls two values from the operand table, performs the desired operation, and returns one value (the result) to the operand table.

Format: $expr_1$ operator $expr_2$

Bracket

Square brackets ([]) are used to delimit blocks of program code in IF and ON operations.

Buffer

A buffer is a section of computer memory dedicated to some particular task. Thus there may be disc buffers which hold data retrieved from the disc, or variable buffers which hold the values of variables during program execution. OPUS utilizes many others besides these.

Byte

A byte is a group of bits (binary digits), each bit having a value of zero (off) or one (on). Eight-bit microprocessors utilize bytes (or words) of 8 bits. One byte will hold one ASCII character, or part of a number.

Character

A character is defined as the ASCII representation of a decimal number from Ø to 255. One character will require one byte of memory.

Comment

Non-executable comments may be entered in source programs using the REM statement.

Constant

A constant is defined as either a literal string or a numerical value. Some examples of constants follow:

```
3.555
-.009
"THIS IS A STRING"
```

Control Characters

Control characters are special ASCII characters dedicated to a particular editing or controlling feature.

Control H (or underline character)	Deletes the last character entered in either command mode or a program INPUT statement. Control H will backspace the cursor one character. The underline character will type an underline after the character. This control character may be typed in repeatedly, each time deleting the corresponding character previously entered.
Control X	Types a backslash (\) and deletes the entire line of input entered either in command mode or in an INPUT statement. OPUS will ignore the line altogether and the user may re-enter the correct line.
Control C	May be used to interrupt program execution, either during an INPUT statement or while the program is running.
Control S	Suppresses all output until a Control Q is typed.
Control Q	Releases output. It is ignored if Control S was not previously typed.

Crash

Should the computer "crash" (i.e., should OPUS become inoperable due to hardware or software failure), the user will lose the program in memory and must re-boot the system. If a crash occurs during an operation that changes data on the disc, it is possible that the disc may become unusable. If it does, the only option left to the user is to re-format the disc in the System Generation Routine.

Delimiters

Delimiters may be defined as special charaters that set the boundaries of particular operations or procedures. For example, square brackets delimit (define the boundaries of) a block.

Editing

Editing source programs in OPUS is much the same as in any BASIC language. Each line entered must be preceded by a line number from 1 to 9999. All source lines must be edited in the command mode. The following procedures are available:

1. Entering New Code: A line of code is entered simply by typing a line number, followed by the line of program code. As many operations as desired may be entered on one line; the only restriction is that the line must be no longer than the length of the input line, as specified during the system initialization routine. Lines will be automatically inserted in the program by OPUS according to the numerical value of the line number. Lines may be entered in any order and OPUS will determine where they go by the numerical value of each line number.

2. Replacing Code: Should a line have to be changed, the programmer need but enter the same line number followed by the correct code; OPUS will automatically replace the old code by the new. The entire line must be re-entered -- it is not possible to change only a portion of the line.

3. Deleting Code: If only one line of program code needs to be deleted, the programmer should type the line number associated with the line and hit the RETURN key. The line will immediately be deleted from the program. If more than one line of program code needs to be deleted (all in the same numerical sequence), the programmer should use the DELete statement.

4. Renumbering Lines: Many times the programmer will not allow enough gap between line numbers to be able to insert another line of code. If this happens, the RENumber command may be used to renumber the entire program, or just one section, in intervals far enough apart to allow further insertions.

It should be kept in mind that, in command mode, OPUS/ONE determines whether a line is being edited or an operation is being entered by determining if the first character input is a number. If the first character is a number, it is assumed that a program is being edited. Otherwise, OPUS/ONE will assume that an operation is being entered which is to be executed immediately. All spaces in a line are ignored by OPUS/ONE (unless contained within a literal string).

Execute

A program is said to be executing when it is in the RUN mode, i.e., OPUS is performing all operations specified by the user in the program.

Expression

An expression may be defined as a constant, a variable, or one or more operations which produce a value (return a value to the operand table). Following are some examples of expressions:

```
99.9                    Z=34
"GOOGLE"                (BB IS NOT CC)
CAT                     X<"HORSE"
A*B-C
```

Floppy Disc

Floppy discs are mass storage devices that may permanently store programs and data files, by means of a head reading or changing magnetic flux on a diskette.

Infinite Loop

Program execution is in an infinite loop if it is looping continuously through a section of code with no way of terminating. Unless the BReaK function is enabled, the user may strike Control C to interrupt the loop. However, if interrupts have been disabled by the BReaK function, it will be necessary to HALT the computer and follow the restart procedures.

Interrupts

A program or operation may be interrupted at any time by striking Control C from the current input device. This may be overridden only by setting the BReaK command to a non-zero number. When an interrupt is received, control will return to the command mode. Some operations must finish execution before the interrupt is acknowledged. A Control C also will interrupt a program during an INPUT statement. OPUS does not utilize the vectored interrupt system. Instead it scans for interrupts from the current input device.

Label

A label is a literal string within a program to which a statement refers. A label may simultaneously be an operand of another statement. It may be of any length (up through 127 characters) and may contain any ASCII character; the label must be enclosed within quotes.

Line Numbers

Line numbers are used for editing (adding, changing, and deleting lines of code) a source program. The range allowed for line numbers is from 1 to 9999, and all must be integers. When a source program is compiled, line numbers are ignored and will not appear in the object code. Therefore no statements will ever refer to line numbers.

List

A list is defined as one or more constants, variables, or expression values that are optionally separated by commas. Several commands will require lists of parameters. Example of a list:

```
-78, CAT, 5*66/4, "HI THERE", X
```

List Delimiter

The comma is the most frequently used delimiter to separate variables, constants, and expressions within a list. However, the comma is not necessary if the value types are different. For example:

```
PRINT A, "HI", 36
```

can be entered as:

```
PRINT A "HI" 36
```

Literal String

A literal string is defined as one or more ASCII characters enclosed within quotes. The maximum length is 127 characters. Here are two literal strings:

```
"THIS IS A LITERAL STRING"
"5.4532"
```

Memory Requirements

The following considerations should be taken into account when determining how much memory is needed to satisfactorily run OPUS:

1. OPUS/ONE cassette/paper tape version requires 16K for the operating system, OPUS/ONE disc version requires 20K, and OPUS/TWO, 24K.

2. Memory requirements increase correspondingly with greater number precision. The number routines require a buffer of 50 bytes for 6-digit precision; for 126 digits of precision, the buffer utilizes 650 bytes.

3. The sizes of the operand table, the constant table and the variable name table are all user-determined during system generation, and may be shortened or lengthened depending upon memory restrictions and program requirements.

4. The variable value buffer may be controlled by reusing variables in the program -- the more variables used, the more memory required. Matrices, in particular, will consume memory.

5. The user determines how many disc files are assigned at one time within a program. Each file requires about 160 bytes of memory.

6. Each disc drive on the system will require about 412 bytes of main memory.

7. User-defined I/O drivers will add to the memory requirements.

8. Keep in mind that both a source program and its corresponding object code must reside in memory when the former is compiled. However, the object code may be saved, the source deleted, and the object reloaded to reside in memory alone for execution.

9. If a source program cannot be compiled without memory overflow, it can be compiled in sections, i.e., sections may be saved as object programs and linked together for execution.

Number-to-String

Any operation that requires string values, such as LEN and & will automatically make sure that the argument is in string format by converting any numerical value to a string prior to the operation. Every digit in the number (including decimal point and sign) will be changed to the corresponding ASCII character. If the value in question is a variable, the converted value form will not replace the variable value but will be used only temporarily for the operation.

Object Program

The object program is produced from the source program with the COMpile command. The following comments about object programs explain the differences between them and source programs:

1. The object program contains no line numbers or line separations, but is treated as one continuous line of operators and operands.

2. All operations are stored in postfix notation -- the operators follow their operands; i.e., A-B*C would be stored as ABC*-. The source, however, is stored in the order in which it is entered.

3. The conversion to postfix notation requires that all statements be given a priority number. This priority determines the order of execution. All parentheses and commas are deleted in the object code because of this priority order. Usually, in other languages, only mathematical expressions are converted in this manner, but in OPUS, the entire program is compiled into this notation. See the Statement Table in Appendix D. for the priority order.

4. The object program may not be listed or edited in any manner.

5. The object program may be stored with the CSAVE command and retrieved with the LOAD command.

6. The object program is the program that actually is executed. No source program may be run without first compiling it into object code.

Operand

An operand is a constant or variable value that is used by a statement during the particular operation. The operand will be in number, string or matrix (combination of numbers and strings) format. Expression values and arguments are often referred to as operands.

Operation

An operation is defined as one statement with its required argument(s). The following is an operation:

```
PRINT "HI THERE"
```

Operator

An operator is a statement or instruction.

Parentheses

Parentheses are used in three instances:

1. To give priority to specific operations. The operation within the parentheses will be executed before those outside. In this example, though it normally has a lower priority, subtraction is executed before multiplication:

```
3*(4-2)=6  Where as:  3*4-2=10
```

And in the next example, the IF statement is executed prior to the PRINT statement:

```
PRINT (IF X IS "CAT" [THEN "MEOW"])
```

2. As matrix, or matrix element, delimiters. Parentheses must enclose the element positions in each dimension.

```
M(3,4)
DIM XX(5,5,5,5 )
```

3. As substring delimiters. If a substring (part of a string) is referenced.

the starting and ending character positions must be enclosed by parentheses.

```
PRINT S$(3,4)
Z = X$(6) & Z$(1,D)
```

Patches

A patch is a revised or new section of machine code incorporated in the operating system. The user of OPUS may conveniently make I/O and disc driver patches (during system generation).

Peripheral Device

This is any I/O device (disc, cassette, terminal, etc.) which communicates with the central processing unit.

Postfix Notation

Postfix notation refers to a method of representing expressions in which operands precede the operator. Normally, expressions are written with infix notation, i.e., the operators are between the operands. This works well when it is possible to look at an expression and immediately see which operations should be considered first. However, the computer has not yet developed to the point where it can glance at an expression and make such decisions easily. By putting the expression into postfix notation, it becomes a simple task to scan the expression from left to right and operate accordingly. Some examples of the two types of notation follow:

Infix Notation	Postfix Notation
A-B	AB-
5-(X-Y)	5XY--
(2.3-H)/X*Y	2.3H-X/Y*
PRINT A,B	AB PRINT
LOOP I, 1 TO 5	I 1 LOOP 5 TO

As the computer scans the expression, it pushes all operands (constants or variables) onto a stack (operand table). When an operator is reached, it pulls the required operands from the stack, performs the operation and optionally returns a value to the stack. Postfix notation is dependent upon the priorities of every operator. Higher priority statements must be executed before lower priority statements and will, therefore, appear first in postfix notation. As can be seen from the examples, parentheses are not needed in postfix expressions due to this priority structure.

In OPUS, not only mathematical expressions are converted to postfix notation, but also all statements and commands. In fact, the entire source program is translated into this notation and is then called the object program. It should be noted also that source programs may actually be written in postfix notation and be accepted by OPUS as valid code.

Priority Structure

Every statement in OPUS is given a priority value to determine the order of execution. A statement with higher priority will be executed before one with lower priority. If the priority is the same for two operations, they will be executed in the order that they are entered.

The user is no doubt familiar with this concept, when applied to mathematical operations: multiplication has higher priority than addition, the same priority as division, and lower priority than exponentiation. For example: 3*4-2 is 10 (multiplication operated on before subtraction); 3-4*2 is -5 (multiplication still higher priority). If subtraction is to take priority, parentheses are required: 3*(4-2) is 6. In OPUS, this concept applies to all statements. Therefore, parentheses must be used to force execution of a lower priority command before a higher priority command. For the following examples, remember that assignment normally has higher priority than the conditional IF statement.

```
A= (IF X [THEN 3] ELSE [THEN 4])
```

Because parentheses enclose the conditional statement, it is executed first. Thus, this example assigns either 3 or 4 to A, depending upon whether X is true or false. If parentheses are left out, an assignment error will result because there is no value to assign to A.

```
A = IF X [THEN 3] ELSE [THEN 4]
= ERROR
```

See Statement Table for the priority values for all OPUS statements. A higher number designates a higher priority. It is the relative difference that is important to remember -- not the actual values of each.

Program

A program is a group of instructions given to the computer to solve a problem. In OPUS, the programmer enters the source program, which is then compiled into the object program. The latter may then be executed (RUN); and theoretically, it produces a solution to the problem.

Program Code

Program code is defined as the statement, constants and variables composing a program.

Program Counter

This is a pointer in OPUS giving the location within the program of the statement currently being executed.

PROM

PROM is an abbreviation for Programmable Read-Only-Memory. Machine code programmed into PROM may be executed at any time, but normally may not be changed under program control. PROM is non-volatile, meaning that the contents of memory will not be affected when the power is shut off. A great deal of time and effort is saved if the OPUS loader is put in PROM or ROM. Instead of having to enter the loader byte by byte through the front panel switches, one need only examine the starting location of the PROM and RUN.

Protected Memory

Protected memory is memory which may not be written on or changed. PROM and ROM are examples of protected memory. The term may also refer to memory locations that are outside of the boundaries of memory in the current system configuration.

Quotation Marks

Quotation marks are used to delimit literal strings.

RAM

RAM is an abbreviation for Random Access Memory. This is memory that may be read from or written into randomly at any specified location. It may be either volatile or non-volatile.

Restart

If a program is in an infinite loop that cannot be interrupted, or some malfunction has occurred causing OPUS to be inoperable, the system may be manually restarted. This will force control back to the start of OPUS. The following steps should be taken:

1. HALT the computer.
2. EXAMINE memory at location ØØØ 1ØØ octal. (ØØ4Ø hex)
3. Make sure that the sense switches are set to the correct device number (see Bringing Up Initialized OPUS, Section I.E.).
4. Push RUN.

OPUS should return to command mode and print the heading and DAY? If this does not happen, it is most likely that memory data has been destroyed, and the user will have to reload OPUS from the beginning; program and table buffers will be cleared.

ROM

ROM is an abbreviation for Read-Only-Memory. The description under PROM also applies to ROM.

Source Program

The source program is the program that the user enters. OPUS abbreviates the actual ASCII code as it comes in by converting all numbers to number format, strings to string format, commands and statements to numbers, and deleting all spaces. A source program may be LISTed and edited, but not executed until it has been COMpiled into the object program.

Spaces

All spaces entered from the input device that are not contained in a string will be deleted immediately if in the command mode. Spaces will not be deleted from data entered in response to an INPUT statement.

Statement

A statement is a general term to describe all commands, functions, and unary or binary operators. Any reserved symbol or word which is a direct instruction to the computer is called a statement. Note: No spaces may be imbedded within a statement.

Statement Delimiter

Since multiple statements may go on one line in a program, they need to be separated by a symbol. The semicolon is a convenient statement delimiter, though it is in itself a valid operation (see ;). It should be stressed that OPUS does not check for any statement separator, but will take the statements as they come and execute them according to priority.

String-to-Number

All statements requiring an argument in number format will convert all string arguments to number form prior to the operation. OPUS determines this number form by scanning the string from left to right until it reaches either a non-numerical character or the end of the string. If the first character in the string is not a numerical character, the number returned will be Ø. A numerical character may be any of the following: a digit (from Ø to 9), "-", "+", ".", or "E" (exponential format). Following are some examples of strings converted to numbers:

```
"12"        becomes 12
"-2.5"      becomes -2.5
"CAT"       becomes Ø
```

"3DOGS" becomes 3
"3.5E2" becomes 350
"+10" becomes 10

If the value in question is a variable, the converted value form will not replace the variable value, but will be used only temporarily for the operation.

Terminals

Terminals are input and/or output devices which send and recieve data to and from the computer under user control. Terminals operating with OPUS must send and receive ASCII characters.

Unary Operator

A unary operator is a statement that requires one argument and returns one value. Perhaps the most common unary operator is the negation sign (-).

Value Format

Every value of an operand is either stored in number format or string format. The former is a BCD (Binary Coded Decimal) floating point number and the latter, a string of ASCII characters.

Variables

A variable is a name given to a numerical value, a string value or a matrix. The following rules apply to variable use:

1. The variable name must consist of upper case alphabetical characters only. No digits or other ASCII characters may be imbedded within the name. The name is not delimited by quotes.

2. The name may not contain any reserved OPUS statement word. Thus the variable CAT is valid, but HORSE contains the reserved statement OR, and therefore is invalid.

3. The length of the variable name must be less than 128 characters.

4. A variable may have the following values:

   Number (any negative or positive number)
   String (any sequence of ASCII characters)
   Matrix (determined by a DIM statement)

5. The variable value may, at any point in the program, be changed from a number to a string, or a string to a number.

6. A simple variable (string or number value) may be changed at any time into a matrix variable by using the DIM statement. However, once a variable has a matrix value, it cannot be changed back to a simple variable.

7. If a statement using a variable requires a different format (number/-string), the value will be temporarily converted to the format required. However, upon completion of the operation, the variable will have the same value in the same format as that prior to the operation.

XVII. APPENDICES

A. STANDARD DRIVERS

This Appendix lists all standard drivers that may be used with each version of OPUS under the System Generation Routine.

The three sections listed here give the drivers for the serial I/O devices, the port initialization routines, and the disc driver routines. Under each section is listed the mnemonics for each available driver, applicable hardware specifications, and the machine code listings for the drivers.

Serial I/O Interface Routines

The code is the switch code determining the interface with which the system generation is to communicate (defined in Section I. C.).

Code	Mnemonic	Interface
I1	MS	MITS 2SIO interface board
I2	MT	MITS SIOC teletype board
I3	IA	IMSAI SIO, channel A
	IB	IMSAI SIO, channel B
	MP	MITS 4PIO parallel board
	MC	MITS ACR board

The following table gives the channel specifications for each interface; the heading definitions are:

Status — Lowest port number + status = status port number
Data — Lowest port number + data = data port number
Inp Bit — Bit position in the status byte for "data byte received"
Active — Ø = Inp Bit low when data received and 1 = Inp Bit high when data received
Out Bit — Bit position in the status byte for "okay to send"
Active — Ø = Out Bit low when okay to send and 1 = Out Bit high when okay to send

Mnemonic	Status	Data	Inp Bit	Active	Out Bit	Active
MS	Ø	1	Ø	1	1	1
MP	Ø	1	6	1	7	1
MC	Ø	1	Ø	Ø	7	Ø
MT	Ø	1	Ø	Ø	7	Ø
IA	3	2	1	1	Ø	1
IB	5	4	1	1	Ø	1

Driver Listings

MITS 2SIO

```
Input Subroutine:       LOOP    IN Ø        Loop until data received
                                RRC
                                JNC LOOP
                                IN 1        Input data byte
                                RET

Output Subroutine:      LOOP    IN Ø        Loop until OK to send
                                ANI 2
                                JZ LOOP
                                MOV AB      Move byte in B to Register A
                                OUT 1       Output data byte
                                RET

Interrupt Subroutine:           IN Ø        Input from status & check for data
                                RRC
                                RET
```

MITS 4PIO

```
Input Subroutine:       LOOP    IN Ø        Loop until data received
                                ANI 1ØØ
                                JZ LOOP
                                IN 1        Input data byte
                                RET

Output Subroutine:      LOOP    IN 2        Loop until OK to send
                                ANI 2ØØ
                                JZ LOOP
                                MOV AB
                                OUT 3       Output byte
                                NOP
                                IN 3        Clear input
                                RET

Interrupt Subroutine:           IN Ø        Input status
                                ANI 1ØØ
                                RAL
                                RAL
                                RET
```

MITS SIOC

```
Input Subroutine:       LOOP    IN Ø        Loop until data received
                                RRC
                                JC LOOP
                                IN 1
                                RET
```

```
Output Subroutine:      LOOP    IN Ø        Loop until OK to send
                                RLC
                                JC LOOP
                                MOV AB
                                OUT 1       Output byte
                                RET

Interrupt Subroutine:           IN Ø        Input status
                                RRC
                                CMC
                                RET
```

MITS ACR

```
Input Subroutine:       LOOP    IN Ø        Loop until data received
                                RRC
                                JC LOOP
                                IN 1        Input data byte
                                RET

Output Subroutine:      LOOP    IN Ø        Loop until OK to send
                                RLC
                                JC LOOP
                                MOV AB
                                OUT 1       Output byte
                                RET

Interrupt Subroutine:           IN Ø        Input status
                                RRC
                                CMC
                                RET
```

IMSAI SIO Channel A

```
Input Subroutine:       LOOP    IN 3        Loop until data received
                                ANI 2
                                JZ LOOP
                                IN 2        Input data
                                RET

Output Subroutine:      LOOP    IN 3        Loop until OK to send
                                ANI 1
                                JZ LOOP
                                MOV AB
                                OUT 2       Output byte
                                RET

Interrupt Subroutine:           IN 3        Input status
                                RRC
                                RRC
                                RET
```

IMSAI SIO Channel B

```
Input Subroutine:       LOOP    IN 5        Loop until data received
                                ANI 2
                                JZ LOOP
                                IN 4        Input data
                                RET

Output Subroutine:      LOOP    IN 5        Loop until OK to send
                                ANI 1
                                JZ LOOP
                                MOV AB
                                OUT 4       Output byte
                                RET

Interrupt Subroutine:           IN 5        Input status
                                RRC
                                RRC
                                RET
```

Port Initialization Routines

Mnemonic	Interface
MS	MITS 2SIO
MT	MITS SIOC teletype
IS	IMSAI SIO (both channels A and B)
MC	MITS ACR cassette
MP	MITS 4PIO parallel

Driver Listings

MITS 2SIO

```
MVI A,3       Master reset
OUT Ø
MVI A,25      Clock: /16, 8 data bits, 1 stop bit, no parity
OUT Ø
RET
```

MITS SIOC

```
IN 1          Clear data
RET
```

MITS ACR

```
IN 1          Clear data
RET
```

MITS 4PIO

```
XRA A
OUT Ø
NOP
OUT 1
NOP
OUT 2
CMA
OUT 3
MVI A,6
OUT 2
MVI A,56
OUT 2
RET
```

IMSAI SIO

```
MVI A,201
OUT 3
MVI A,100
OUT 3
MVI A,112
OUT 3
MVI A,47
OUT 3
XRA A
OUT 2
MVI A,201
OUT 5
MVI A,100
OUT 5
MVI A,112
OUT 5
MVI A,47
OUT 5
XRA A
OUT 4
RET
```

Disc Drivers

Mnemonic	Interface	Bytes/Sector	Sectors/Track	Tracks/Disc
MD	MITS Altair drive	128	32	77
IC	iCOM drive	128	26	77

Driver Listings

NOTE: The format of the code listed below is according to the specifications of A.S.I.'s 8080 Assembler.

```
*******  ALTAIR DISC DRIVER  *******

*---WRITE SECTOR---*

<MW RITE>       MVI B,10;

<MW RITE Q>     PUSH H;         PUSH D;         PUSH B;

        *-ENABLE,POSITION HEADS-*

                CALL MRENABLE;  CALL MRTRK;

        *-SET UP SECTOR FORMAT-*

                PUSH H;         MOV CL;         MOV AH;
                LXI H,MRBUF;    MVI M,303;      INX H;          MOV MC;
                INX H;          MOV MA;         INX H;          XRA C;
                MOV MA;         INX H;          PUSH H;         LXI B,200;

        *-TRANSFER DATA TO MRBUF BUFFER & CALCULATE CHECKSUM-*

<MW RITE A>     INX H;          LDAX D;         MOV MA;         INX D;
                XRA B;          MOV BA;         DCR C;          JNZ MWRITEA;

                POP H;          MOV MB;         POP H;
                MOV AH;         MVI B,200;      CPI .43;
                JC MWRITEB;     MVI B,300;

<MW RITE B>     DCR L;          MOV AL;         ANI 37;         MOV LA;
                PUSH B;

                PUSH H;         LHLD PDIS;      DCX H;          MOV AM;
                ORA A;          POP H;          JNZ MWRITEBA;

        *-READ PREVIOUS SECTOR (NO VALIDATION)-*
```

```
                CALL MRSEC;     JMP MWRITEBD;

        *-READ AND VALIDATE PREVIOUS SECTOR-*

<MW RITE BA>    MVI C,4;        LXI D,MWBUF;    CALL MRPART;
                JZ MWRITEZ;

<MW RITE BD>    MOV AL;         INR A;          ANI 37;         MOV LA;
                POP B;
                MVI H,1;        MVI C,206;      LXI D,MRBUF;    XCHG;

        *-WAIT FOR CORRECT SECTOR-*

<MW RITE BB>    IN 1;           RRC;            JNC MWRITEBB;
<MW RITE BC>    IN 1;           RRC;            JC MWRITEBC;
                ANI 37;         CMP E;          JNZ MWRITEZ;

        *-OUTPUT BUFFER TO DISC-*

                MOV AB;         OUT 1;          MOV EM;         INX H;

<MW RITE C>     IN 0;           ANA D;          JNZ MWRITEC;
                ADD E;          OUT 2;          MOV AM;         INX H;
                DCR C;          MOV EM;         INX H;          JZ MWRITED;
                DCR C;          OUT 2;          JNZ MWRITEC;

        *-GO TO READ ROUTINE FOR VERIFICATION-*

<MW RITE D>     POP B;          POP D;          POP H;          JMP MRREAD;

        *-FAILED TO WRITE; TRY AGAIN-*

<MW RITE Z>     POP B;          POP B;          POP D;          POP H;
                DCR B;          JNZ MWRITEQ;

        *-FAILURE-*

                MVI A,10;       OUT 1;          SUI 6;          RET;

        *-BUFFER TO HOLD PREFIX OF PREVIOUS SECTOR-*

<MW BUF>        BUF 4;

*---READ SECTOR---*

<MR READ>       MVI B,10;

<MR READ A>     PUSH H;         PUSH D;         PUSH B;         PUSH D;

        *-ENABLE,GET TO TRACK-*

                CALL MRENABLE;  CALL MRTRK;     LXI D,MRBUF;    MVI C,205;

        *-READ SECTOR-*
```

```
                CALL MRPART;    MVI A,10;       OUT 1;          JZ MRREADZ;

        *-SECTOR GOOD: TRANSFER TO SPECIFIED BUFFER-*

                XCHG;           INX H;          POP D;          PUSH H;
                LXI B,200;

<MR READ G>     INX H;          MOV AM;         STAX D;         INX D;
                XRA B;          MOV BA;         DCR C;          JNZ MRREADG;

                POP H;          CMP M;
                POP B;          POP D;          POP H;          JNZ MRREADZA;

        *-CHECKSUM OK, RETURN A=0 TO OPUS-*

                XRA A;          RET;

        *-NOT READ CORRECTLY, TRY AGAIN-*

<MR READ Z>     POP D;          POP B;          POP D;          POP H;

<MR READ ZA>    DCR B;          JNZ MRREADA;

        *-FAILURE, RETURN A=2 TO OPUS-*

                MVI A,2;        RET;

*---READ GIVEN # BYTES FROM SECTOR---*

<MR PART>       MVI B,10;

<MR PART A>     CALL MRSEC;     PUSH B;         PUSH D;

<MR PART B>     IN 0;           RAL;            JC MRPARTB;
                IN 2;           STAX D;         INX D;          DCR C;
                NOP;            JZ MRPARTC;     DCR C;          IN 2;
                STAX D;         INX D;          JNZ MRPARTB;

<MR PART C>     XCHG;           POP H;          PUSH H;

        *-VERIFY PREFIX-*

                MOV AM;         CPI 303;        JNZ MRPARTZ;    INX H;
                MOV AM;         CMP E;          JNZ MRPARTZ;    INX H;
                MOV AM;         CMP D;          JNZ MRPARTZ;    INX H;
                MOV AE;         XRA D;          CMP M;          JNZ MRPARTZ;

                XCHG;           ORA B;
                POP B;          POP B;          RET;

<MR PART Z>     XCHG;           POP D;          POP B;
                DCR B;          JNZ MRPARTA;
```

```
        *-FAILURE-*

                PUSH H;         PUSH D;         PUSH B;
                LXI H,MRP;      LDA MRD;        MOV CA;         MVI B,0;
                DAD B;          MVI M,0;        POP B;
                POP D;          POP H;          JMP MRTRKZA;

*---GET TO TRACK---*

<MR TRK>        MVI A,4;        OUT 1;
                PUSH D;         PUSH H;         LXI H,MRP;
                MVI B,0;        DAD B;          MOV BM;         XTHL;

        *-CHECK FOR TRACK 0/SECTOR 0-*

                MOV AL;         ORA H;          JZ MRTRKZ;

                MOV AB;         SUB H;          MVI E,2;
                JNC MRTRKA;     CMA;            INR A;          DCR E;

<MR TRK A>      ORA A;          MOV DA;         MOV AH;
                XTHL;           JZ MRTRKD;
                MOV MA;

        *-STEP CORRECT # TRACKS-*

<MR TRK B>      IN 0;           ANI 2;          JNZ MRTRKB;
                MOV AE;         OUT 1;          DCR D;          JNZ MRTRKB;

<MR TRK D>      POP H;          POP D;          RET;

        *-MOVE HEADS TO TRACK 0-*

<MR TRK Z>      XTHL;           MVI M,0;        POP H;          POP D;

<MR TRK ZA>     IN 0;           ANI 2;          JNZ MRTRKZA;
                MVI A,2;        OUT 1;
                IN 0;           ANI 100;        JNZ MRTRKZA;
                RET;

*---BUFFER HOLDING TRACK LOCATIONS FOR EACH DRIVE---*

<MRP>           BUF 10;

<MRD>           BUF 1;          *-CURRENT DISC

*---SELECT AND ENABLE DISC---*

<MR ENABLE>     MOV AC;         OUT 0;          STA MRD;
                IN 0;           CPI 377;        JZ MRENABLE;
                RET;

*---GET TO TRUE SECTOR---*
```

```
<MR SEC>        IN 0;           ANI 4;          JNZ MRSEC;

<MR SEC A>      IN 1;           RAR;            JC MRSECA;
                ANI 37;         CMP L;          JNZ MRSECA;     RET;

*---DISC BUFFER---*

<MR BUF>        BUF .138;



******  ICOM DISC DRIVER  ******

*---WRITE SECTOR---*

<IW RITE>       PUSH D;         PUSH H;         MVI A,201;      CALL IRLOOP;
                MVI B,.128;

        *-TRANSFER DATA TO SHIFT REGISTER-*

<IW RITE A>     LDAX D;         OUT 1;          MVI A,61;       CALL IRLOOP;
                DCR B;          INX D;          JNZ IWRITEA;

        *-ENABLE AND LOCATE HEADS-*

                CALL IRTRK;     MVI C,10;

        *-WRITE SECTOR-*

<IW RITE B>     MVI A,5;        CALL IRLOOP;    MVI A,7;        CALL IRLOOP;
                IN 0;           ANI 10;         JZ IWRITEC;

        *-FAILURE TO WRITE, TRY AGAIN-*

                MVI A,13;       CALL IRLOOP;    DCR C;          JNZ IWRITEB;

        *-FAILURE, RETURN A=2-*

                POP H;          POP D;          MVI A,2;        RET;

        *-SUCCESS, RETURN A=0-*

<IW RITE C>     POP H;          POP D;          XRA A;          RET;


*---READ SECTOR---*

<IR READ>       PUSH D;         PUSH H;         MVI A,201;      CALL IRLOOP;

        *-ENABLE, LOCATE HEADS-*

                CALL IRTRK;     MVI C,10;
```

```
        *-READ INTO SHIFT REGISTER-*

<IR READ A>     MVI A,3;        CALL IRLOOP;    IN 0;           ANI 10;
                JZ IRREADB;

        *-FAILURE, TRY AGAIN-*

                CALL IRFLAG;    DCR C;          JNZ IRREADA;

        *-FAILURE, RETURN A=2-*

                POP H;          POP D;          MVI A,2;        RET;

        *-MOVE DATA TO MEMORY-*

<IR READ B>     MVI A,100;      OUT 0;          IN 0;           STAX D;
                MVI C,.128;

<IR READ C>     CALL IRLOOP+2;  INX D;          DCR C;          JZ IRREADD;
                MVI A,101;      OUT 0;          IN 0;           STAX D;
                JMP IRREADC;

        *-SUCCESS, RETURN A=0-*

<IR READ D>     POP H;          POP D;          XRA A;          RET;


*---BUFFER TO HOLD TRACK LOCATIONS---*

<IR TRACK>      BUF 10;

*---ENABLE DISC, GET TO TRACK & SECTOR---*

<IR TRK>        PUSH H;         LXI H,IRTRACK;  MVI B,0;        DAD B;
                XTHL;           MOV AC;         RRC;            RRC;
                ORA L;          INR A;          MOV BA;

<IR TRK A>      MOV AB;         OUT 1;          MVI A,41;       CALL IRLOOP;
                IN 0;           ANI 40;         JZ IRTRKB;
                CALL IRFLAG;    JMP IRTRKA;

<IR TRK B>      MOV AL;         ORA H;          MOV AH;         XTHL;
                JNZ IRTRKC;

        *-SEEK TRACK 0,SECTOR 0-*

                MVI A,15;       CALL IRLOOP;    MOV AB;         OUT 1;
                MVI A,41;       CALL IRLOOP;    XRA A;

<IR TRK C>      CMP M;          MOV MA;         POP H;          RZ;
                OUT 1;          MVI A,21;
                CALL IRLOOP;    MVI A,11;       CALL IRLOOP;    RET;

*---LOOP UNTIL READY---*
```

```
<IR FLAG>       MVI A,13;

<IR LOOP>       OUT 0;          SUB A;          OUT 0;

<IR LOOP A>     IN 0;           RAR;            JC IRLOOPA;
                RET;
```

B. LOADERS

Machine code listings and a description of the loader format are given for our standard A.S.I. loaders:

Section 1: Disc loaders for MITS Altair drive and iCOM drive

Section 2: Paper tape loader using Intel Hex format

Section 3: Cassette loader for MITS ACR

All loaders assume the peripheral devices are strapped to a certain port number. These may easily be changed in the loader.

The loaders are all located at 177/ØØØ Octal (7FØØ Hex). This location may be changed by updating all JUMP references within the loader. In the listings, these reference locations are noted by an "@" symbol.

At the end of each loader there is a specific I/O port initialization routine. This is optional. Its presence makes sure that the terminal on which OPUS is to come up is initialized. Refer to the specifications of the interface board to make sure it is being initialized correctly. A sample port initialization routine has been inserted in the loader listings. Check it carefully.

Both Octal and Hex listings are given for each loader to accomodate all user preferences.

The method in which OPUS resides on each medium is described with each section. Special loaders may be written following these guidelines.

Disc Loaders

The MITS loader assumes that the disc drive has been strapped for ports 8, 9 and 1Ø (1Ø, 11 and 12 Octal).

The iCOM loader assumes that the disc drive has been strapped for ports 192, and 193 (CØ and C1 Hex).

If the loader is to be burned into ROM or PROM memory, because this memory normally runs slower than RAM, the loader portion must be first shifted into RAM prior to execution. A short routine should be inserted preceding the loader to accomplish this shift. The addresses in the loader should remain the same.

Shift Routine:

```
LXI H,LOADER          Set HL to point to loader location
LXI D, 77ØØ Octal     Set DE to start RAM location
          or
       7FØØ Hex
MVI C,255 Decimal     C = number of bytes to shift
```

```
< LOOP >    MOV AM          Move byte from PROM to Register A
            STAX D          Store Register A in RAM
            INX H           Increment HL Register pair
            INX D           Increment DE Register pair
            DCR C           Decrement byte counter
            JNZ LOOP        If more, continue to shift
            JMP 77ØØ Octal  Jump to start of RAM for execution
                  or
                7FØØ Hex

< LOADER >                  Insert normal loader at this point
```

The following specifications describe the format of OPUS as it resides on the disc. This should be studied carefully if the user needs to write a loader for a non-standard disc drive. Both OPUS with the System Generation Routine and the initialized version of OPUS will be dumped on the disc in this format.

1. OPUS resides on the disc sequentially as it appears in memory. The number of bytes of OPUS in each sector is the maximum number of data bytes per sector, minus one (normally 127).

2. Track Ø, Sector Ø holds the first block of OPUS. The sector number is incremented by two for the next block. After the even numbered sectors are full, Sector 1 will be specified, followed by the rest of the odd sectors. After a track is full, the track number will increment by one and the sector number starts over with Sector Ø.

3. The first data byte of each sector is an end-of-file mark. It will be 1 (one) if more sectors follow or a Ø (zero) if the sector is the last sector of OPUS. The rest of the bytes of each sector contain OPUS code.

4. The loader should read in each sector, check for a bad read (checksum), shift the OPUS data bytes sequentially into memory, and finally check to see if this sector is the last one. If not, it should increment the sector and/or track numbers and continue to read.

PAGE	LOC	LABEL	CODE	MNEMONIC		PAGE#: 1

```
*****  NEW LOADER FOR ALTAIR DRIVES (EFFECTIVE 12/12/77)  *****

177   000     <LOAD>          061       LXI   S,LOAD
177   001                     000 @
177   002                     177
177   003                     257       XRA   A
177   004                     323       OUT   .8
177   005                     010
177   006                     333       IN    .8
177   007                     010
177   010                     376       CPI   .255
177   011                     377
177   012                     312       JZ    LOAD
177   013                     000 @
177   014                     177

*-LOAD HEADS AND GET TO TRACK 0-*

177   015                     076       MVI   A,4
177   016                     004
177   017                     323       OUT   .9
177   020                     011
177   021                     056       MVI   L,2
177   022                     002
177   023     <LOADA>         315       CALL  STEP
177   024                     233 @
177   025                     177
177   026                     333       IN    .8
177   027                     010
177   030                     346       ANI   .64
177   031                     100
177   032                     302       JNZ   LOADA
177   033                     023 @
177   034                     177
177   035                     001       LXI   B,0
177   036                     000
177   037                     000
177   040                     021       LXI   D,0
177   041                     000
177   042                     000

*-READ SECTOR AT TRACK B, SECTOR C, DUMP TO MEMORY AT DE-*

177   043     <LOADB>         333       IN    .9
177   044                     011
177   045                     037       RAR
177   046                     332       JC    LOADB
177   047                     043 @
177   050                     177
177   051                     346       ANI   .31
177   052                     037
```

PAGE	LOC	LABEL	CODE	MNEMONIC	
177	053		271	CMP	C
177	054		302	JNZ	LOADB
177	055		043 @		
177	056		177		
177	057		056	MVI	L,.133
177	060		205		
177	061		325	PUSH	D
177	062	<LOADC>	333	IN	.8
177	063		010		
177	064		027	RAL	
177	065		332	JC	LOADC
177	066		062 @		
177	067		177		
177	070		333	IN	.10
177	071		012		
177	072		022	STAX	D
177	073		023	INX	D
177	074		000	NOP	
177	075		055	DCR	L
177	076		312	JZ	LOADD
177	077		111 @		
177	100		177		
177	101		055	DCR	L
177	102		333	IN	.10
177	103		012		
177	104		022	STAX	D
177	105		023	INX	D
177	106		302	JNZ	LOADC
177	107		062 @		
177	110		177		

-VERIFY PREFIX-

177	111	<LOADD>	341	POP	H
177	112		124	MOV	DH
177	113		135	MOV	EL
177	114		176	MOV	AM
177	115		376	CPI	.195
177	116		303		
177	117		302	JNZ	LOADB
177	120		043 @		
177	121		177		
177	122		043	INX	H
177	123		176	MOV	AM
177	124		271	CMP	C
177	125		302	JNZ	LOADB
177	126		043 @		
177	127		177		
177	130		043	INX	H
177	131		176	MOV	AM
177	132		270	CMP	B
177	133		302	JNZ	LOADB
177	134		043 @		
177	135		177		
177	136		043	INX	H

```
PAGE  LOC        LABEL                CODE        MNEMONIC
----  ---        -----                ----        --------

177   137                             171         MOV   AC
177   140                             250         XRA   B
177   141                             276         CMP   M
177   142                             302         JNZ   LOAD
177   143                             000 @
177   144                             177
177   145                             043         INX   H
177   146                             305         PUSH  B
177   147                             106         MOV   BM
177   150                             043         INX   H
177   151                             116         MOV   CM
177   152                             305         PUSH  B
177   153                             101         MOV   BC
177   154                             016         MVI   C,.127
177   155                             177

*-SHIFT DOWN ONE BYTE & CALCULATE CHECKSUM-*

177   156       <LOADE>               043         INX   H
177   157                             176         MOV   AM
177   160                             022         STAX  D
177   161                             023         INX   D
177   162                             250         XRA   B
177   163                             107         MOV   BA
177   164                             015         DCR   C
177   165                             302         JNZ   LOADE
177   166                             156 @
177   167                             177
177   170                             301         POP   B
177   171                             270         CMP   B
177   172                             312         JZ    LOADF
177   173                             176 @
177   174                             177

*-CHECKSUM ERROR-*

177   175                             166         HLT

*-CHECK FOR END-OF-FILE-*

177   176       <LOADF>               171         MOV   AC
177   177                             267         ORA   A
177   200                             312         JZ    END
177   201                             246 @
177   202                             177

*-INCREMENT SECTOR/TRACK AND RETURN FOR MORE-*

177   203                             301         POP   B
177   204                             014         INR   C
177   205                             014         INR   C
177   206                             171         MOV   AC
177   207                             376         CPI   .32
177   210                             040
177   211                             332         JC    LOADB
```

```
PAGE  LOC        LABEL              CODE      MNEMONIC              PAGE#: 4

177   212                           043 @
177   213                           177
177   214                           016       MVI   C,1
177   215                           001
177   216                           312       JZ    LOADB
177   217                           043 @
177   220                           177
177   221                           015       DCR   C
177   222                           056       MVI   L,1
177   223                           001
177   224                           315       CALL  STEP
177   225                           233 @
177   226                           177
177   227                           004       INR   B
177   230                           303       JMP   LOADB
177   231                           043 @
177   232                           177

*** SUBROUTINE TO STEP ONE TRACK ***

177   233      <STEP>               333       IN    .8
177   234                           010
177   235                           346       ANI   2
177   236                           002
177   237                           302       JNZ   STEP
177   240                           233 @
177   241                           177
177   242                           175       MOV   AL
177   243                           323       OUT   .9
177   244                           011
177   245                           311       RET

***** LOAD COMPLETE- UNLOAD HEADS *****

177   246      <END>                076       MVI   A,.128
177   247                           200
177   250                           323       OUT   .8
177   251                           010

*-INITIALIZE OPTIONAL PORT-*


* THE FOLLOWING IS FOR MITS 2SIO, PORT 16 *

177   252                           076       MVI   A,3
177   253                           003
177   254                           323       OUT   .16
177   255                           020
177   256                           076       MVI   A,.21
177   257                           025
177   260                           323       OUT   .16
177   261                           020
177   262                           303       JMP   0
177   263                           000
177   264                           000
```

```
PAGE  LOC       LABEL                CODE       MNEMONIC          PAGE#: 1


****  NEW LOADER FOR ALTAIR DRIVES (EFFECTIVE 12/12/77)  ****

7F    00        <LOAD>               31         LXI    S,LOAD
7F    01                             00   @
7F    02                             7F
7F    03                             AF         XRA    A
7F    04                             D3         OUT    .8
7F    05                             08
7F    06                             DB         IN     .8
7F    07                             08
7F    08                             FE         CPI    .255
7F    09                             FF
7F    0A                             CA         JZ     LOAD
7F    0B                             00   @
7F    0C                             7F

*-LOAD HEADS AND GET TO TRACK 0-*

7F    0D                             3E         MVI    A,4
7F    0E                             04
7F    0F                             D3         OUT    .9
7F    10                             09
7F    11                             2E         MVI    L,2
7F    12                             02
7F    13        <LOADA>              CD         CALL   STEP
7F    14                             9B   @
7F    15                             7F
7F    16                             DB         IN     .9
7F    17                             08
7F    18                             E6         ANI    .64
7F    19                             40
7F    1A                             C2         JNZ    LOADA
7F    1B                             13   @
7F    1C                             7F
7F    1D                             01         LXI    B,0
7F    1E                             00
7F    1F                             00
7F    20                             11         LXI    D,0
7F    21                             00
7F    22                             00

*-READ SECTOR AT TRACK B, SECTOR C, DUMP TO MEMORY AT DE-*

7F    23        <LOADB>              DB         IN     .9
7F    24                             09
7F    25                             1F         RAR
7F    26                             DA         JC     LOADB
7F    27                             23   @
7F    28                             7F
7F    29                             E6         ANI    .31
7F    2A                             1F
```

PAGE	LOC	LABEL	CODE		MNEMONIC	
7F	2B		B9		CMP	C
7F	2C		C2		JNZ	LOADB
7F	2D		23	@		
7F	2E		7F			
7F	2F		2E		MVI	L,.133
7F	30		85			
7F	31		D5		PUSH	D
7F	32	<LOADC>	DB		IN	.8
7F	33		08			
7F	34		17		RAL	
7F	35		DA		JC	LOADC
7F	36		32	@		
7F	37		7F			
7F	38		DB		IN	.10
7F	39		0A			
7F	3A		12		STAX	D
7F	3B		13		INX	D
7F	3C		00		NOP	
7F	3D		2D		DCR	L
7F	3E		CA		JZ	LOADD
7F	3F		49	@		
7F	40		7F			
7F	41		2D		DCR	L
7F	42		DB		IN	.10
7F	43		0A			
7F	44		12		STAX	D
7F	45		13		INX	D
7F	46		C2		JNZ	LOADC
7F	47		32	@		
7F	48		7F			

-VERIFY PREFIX-

7F	49	<LOADD>	E1		POP	H
7F	4A		54		MOV	DH
7F	4B		5D		MOV	EL
7F	4C		7E		MOV	AM
7F	4D		FE		CPI	.195
7F	4E		C3			
7F	4F		C2		JNZ	LOADB
7F	50		23	@		
7F	51		7F			
7F	52		23		INX	H
7F	53		7E		MOV	AM
7F	54		B9		CMP	C
7F	55		C2		JNZ	LOADB
7F	56		23	@		
7F	57		7F			
7F	58		23		INX	H
7F	59		7E		MOV	AM
7F	5A		B8		CMP	B
7F	5B		C2		JNZ	LOADB
7F	5C		23	@		
7F	5D		7F			
7F	5E		23		INX	H

PAGE	LOC	LABEL	CODE	MNEMONIC	
7F	5F		79	MOV	AC
7F	60		A8	XRA	B
7F	61		BE	CMP	M
7F	62		C2	JNZ	LOAD
7F	63		00	@	
7F	64		7F		
7F	65		23	INX	H
7F	66		C5	PUSH	B
7F	67		46	MOV	BM
7F	68		23	INX	H
7F	69		4E	MOV	CM
7F	6A		C5	PUSH	B
7F	6B		41	MOV	BC
7F	6C		0E	MVI	C,.127
7F	6D		7F		
-SHIFT DOWN ONE BYTE & CALCULATE CHECKSUM-					
7F	6E	<LOADE>	23	INX	H
7F	6F		7E	MOV	AM
7F	70		12	STAX	D
7F	71		13	INX	D
7F	72		A8	XRA	B
7F	73		47	MOV	BA
7F	74		0D	DCR	C
7F	75		C2	JNZ	LOADE
7F	76		6E	@	
7F	77		7F		
7F	78		C1	POP	B
7F	79		B8	CMP	B
7F	7A		CA	JZ	LOADF
7F	7B		7E	@	
7F	7C		7F		
-CHECKSUM ERROR-					
7F	7D		76	HLT	
-CHECK FOR END-OF-FILE-					
7F	7E	<LOADF>	79	MOV	AC
7F	7F		B7	ORA	A
7F	80		CA	JZ	END
7F	81		A6	@	
7F	82		7F		
-INCREMENT SECTOR/TRACK AND RETURN FOR MORE-					
7F	83		C1	POP	B
7F	84		0C	INR	C
7F	85		0C	INR	C
7F	86		79	MOV	AC
7F	87		FE	CPI	.32
7F	88		20		
7F	89		DA	JC	LOADB

```
PAGE   LOC        LABEL                CODE        MNEMONIC          PAGE#: 4

7F     8A                              23  @
7F     8B                              7F
7F     8C                              0E          MVI    C,1
7F     8D                              01
7F     8E                              CA          JZ     LOADB
7F     8F                              23  @
7F     90                              7F
7F     91                              0D          DCR    C
7F     92                              2E          MVI    L,1
7F     93                              01
7F     94                              CD          CALL   STEP
7F     95                              9B  @
7F     96                              7F
7F     97                              04          INR    B
7F     98                              C3          JMP    LOADB
7F     99                              23  @
7F     9A                              7F

*** SUBROUTINE TO STEP ONE TRACK ***

7F     9B          <STEP>              DB          IN     .8
7F     9C                              08
7F     9D                              E6          ANI    2
7F     9E                              02
7F     9F                              C2          JNZ    STEP
7F     A0                              9B  @
7F     A1                              7F
7F     A2                              7D          MOV    AL
7F     A3                              D3          OUT    .9
7F     A4                              09
7F     A5                              C9          RET

***** LOAD COMPLETE- UNLOAD HEADS *****

7F     A6          <END>               3E          MVI    A,.128
7F     A7                              80
7F     A8                              D3          OUT    .8
7F     A9                              08

*-INITIALIZE OPTIONAL PORT-*


* THE FOLLOWING IS FOR MITS 2SIO, PORT 16 *

7F     AA                              3E          MVI    A,3
7F     AB                              03
7F     AC                              D3          OUT    .16
7F     AD                              10
7F     AE                              3E          MVI    A,.21
7F     AF                              15
7F     B0                              D3          OUT    .16
7F     B1                              10
7F     B2                              C3          JMP    0
7F     B3                              00
7F     B4                              00
```

```
PAGE  LOC      LABEL            CODE       MNEMONIC          PAGE#: 1

*****  LOADER FOR ICOM DISC DRIVES  *****

177   000      <LOAD>           061        LXI    S,LOAD
177   001                       000 @
177   002                       177

*-ENABLE DRIVE 0-*

177   003      <LOADA>          076        MVI    A,.129
177   004                       201
177   005                       315        CALL   LOOP
177   006                       157 @
177   007                       177
177   010                       076        MVI    A,.13
177   011                       015
177   012                       315        CALL   LOOP
177   013                       157 @
177   014                       177
177   015                       333        IN     .192
177   016                       300
177   017                       346        ANI    .32
177   020                       040
177   021                       302        JNZ    LOADA
177   022                       003 @
177   023                       177
177   024                       001        LXI    B,1
177   025                       001
177   026                       000
177   027                       041        LXI    H,0
177   030                       000
177   031                       000

*-READ SECTOR AT TRACK B, SECTOR C, DUMP TO MEMORY AT HL-*

177   032      <LOADB>          171        MOV    AC
177   033                       323        OUT    .193
177   034                       301
177   035                       076        MVI    A,.33
177   036                       041
177   037                       315        CALL   LOOP
177   040                       157 @
177   041                       177
177   042                       076        MVI    A,3
177   043                       003
177   044                       315        CALL   LOOP
177   045                       157 @
177   046                       177
177   047                       333        IN     .192
177   050                       300
177   051                       346        ANI    .8
177   052                       010
```

```
177   053                        312       JZ     LOADC
177   054                        057 @
177   055                        177

*-CRC ERROR: HALT-*

177   056                        166       HLT

*-READ BYTES INTO MEMORY-*

177   057     <LOADC>            076       MVI    A,.64
177   060                        100
177   061                        323       OUT    .192
177   062                        300
177   063                        333       IN     .192
177   064                        300
177   065                        127       MOV    DA
177   066                        036       MVI    E,.127
177   067                        177
177   070     <LOADD>            315       CALL   LOOPA
177   071                        161 @
177   072                        177
177   073                        076       MVI    A,.65
177   074                        101
177   075                        323       OUT    .192
177   076                        300
177   077                        333       IN     .192
177   100                        300
177   101                        167       MOV    MA
177   102                        043       INX    H
177   103                        035       DCR    E
177   104                        302       JNZ    LOADD
177   105                        070 @
177   106                        177
177   107                        315       CALL   LOOPA
177   110                        161 @
177   111                        177

*-CHECK FOR END-OF-FILE-*

177   112                        172       MOV    AD
177   113                        267       ORA    A
177   114                        312       JZ     END
177   115                        173 @
177   116                        177

*-INCREMENT SECTOR BY TWO-*

177   117                        014       INR    C
177   120                        014       INR    C
177   121                        171       MOV    AC
177   122                        376       CPI    .27
177   123                        033
177   124                        332       JC     LOADB
177   125                        032 @
```

```
PAGE  LOC        LABEL          CODE       MNEMONIC              PAGE#:

177   126                       177
177   127                       016        MVI    C,2
177   130                       002
177   131                       312        JZ     LOADB
177   132                       032 @
177   133                       177
177   134                       004        INR    B
177   135                       170        MOV    AB
177   136                       323        OUT    .193
177   137                       301
177   140                       076        MVI    A,.17
177   141                       021
177   142                       315        CALL   LOOP
177   143                       157 @
177   144                       177
177   145                       076        MVI    A,.9
177   146                       011
177   147                       315        CALL   LOOP
177   150                       157 @
177   151                       177
177   152                       016        MVI    C,1
177   153                       001
177   154                       303        JMP    LOADB
177   155                       032 @
177   156                       177

*-DELAY LOOP SUBROUTINE-*

177   157      <LOOP>           323        OUT    .192
177   160                       300
177   161      <LOOPA>          227        SUB    A
177   162                       323        OUT    .192
177   163                       300
177   164      <LOOPB>          333        IN     .192
177   165                       300
177   166                       037        RAR
177   167                       332        JC     LOOPB
177   170                       164 @
177   171                       177
177   172                       311        RET

*-OPUS LOADED: OPTIONALLY INITIALIZE PORTS-*

*-THE FOLLOWING INITIALIZATION FOR IMSAI SIO LOW PORT 32-*

177   173      <END>            076        MVI    A,.129
177   174                       201
177   175                       323        OUT    .35
177   176                       043
177   177                       076        MVI    A,.64
177   200                       100
177   201                       323        OUT    .35
177   202                       043
177   203                       076        MVI    A,.74
```

PAGE	LOC	LABEL	CODE	MNEMONIC		PAGE#: 4
177	204		112			
177	205		323	OUT	.35	
177	206		043			
177	207		076	MVI	A,.39	
177	210		047			
177	211		323	OUT	.35	
177	212		043			
177	213		257	XRA	A	
177	214		323	OUT	.34	
177	215		042			
177	216		303	JMP	0	
177	217		000			
177	220		000			

PAGE	LOC	LABEL	CODE	MNEMONIC	PAGE#: 1
***** LOADER FOR ICOM DISC DRIVES *****					
7F	00	<LOAD>	31	LXI S,LOAD	
7F	01		00 @		
7F	02		7F		
-ENABLE DRIVE 0-					
7F	03	<LOADA>	3E	MVI A,.129	
7F	04		81		
7F	05		CD	CALL LOOP	
7F	06		6F @		
7F	07		7F		
7F	08		3E	MVI A,.13	
7F	09		0D		
7F	0A		CD	CALL LOOP	
7F	0B		6F @		
7F	0C		7F		
7F	0D		DB	IN .192	
7F	0E		C0		
7F	0F		E6	ANI .32	
7F	10		20		
7F	11		C2	JNZ LOADA	
7F	12		03 @		
7F	13		7F		
7F	14		01	LXI B,1	
7F	15		01		
7F	16		00		
7F	17		21	LXI H,0	
7F	18		00		
7F	19		00		
-READ SECTOR AT TRACK B, SECTOR C, DUMP TO MEMORY AT HL-					
7F	1A	<LOADB>	79	MOV AC	
7F	1B		D3	OUT .193	
7F	1C		C1		
7F	1D		3E	MVI A,.33	
7F	1E		21		
7F	1F		CD	CALL LOOP	
7F	20		6F @		
7F	21		7F		
7F	22		3E	MVI A,3	
7F	23		03		
7F	24		CD	CALL LOOP	
7F	25		6F @		
7F	26		7F		
7F	27		DB	IN .192	
7F	28		C0		
7F	29		E6	ANI .8	
7F	2A		08		

```
PAGE  LOC      LABEL              CODE        MNEMONIC          PAGE#: 2
----  ---  ----------------      ----  ----------------------

7F    2B                         CA        JZ      LOADC
7F    2C                         2F  @
7F    2D                         7F

*-CRC ERROR: HALT-*

7F    2E                         76        HLT

*-READ BYTES INTO MEMORY-*

7F    2F       <LOADC>           3E        MVI     A,.64
7F    30                         40
7F    31                         D3        OUT     .192
7F    32                         C0
7F    33                         DB        IN      .192
7F    34                         C0
7F    35                         57        MOV     DA
7F    36                         1E        MVI     E,.127
7F    37                         7F
7F    38       <LOADD>           CD        CALL    LOOPA
7F    39                         71  @
7F    3A                         7F
7F    3B                         3E        MVI     A,.65
7F    3C                         41
7F    3D                         D3        OUT     .192
7F    3E                         C0
7F    3F                         DB        IN      .192
7F    40                         C0
7F    41                         77        MOV     MA
7F    42                         23        INX     H
7F    43                         1D        DCR     E
7F    44                         C2        JNZ     LOADD
7F    45                         38  @
7F    46                         7F
7F    47                         CD        CALL    LOOPA
7F    48                         71  @
7F    49                         7F

*-CHECK FOR END-OF-FILE-*

7F    4A                         7A        MOV     AD
7F    4B                         B7        ORA     A
7F    4C                         CA        JZ      END
7F    4D                         7B  @
7F    4E                         7F

*-INCREMENT SECTOR BY TWO-*

7F    4F                         0C        INR     C
7F    50                         0C        INR     C
7F    51                         79        MOV     AC
7F    52                         FE        CPI     .27
7F    53                         1B
7F    54                         DA        JC      LOADB
7F    55                         1A  @
```

```
PAGE  LOC        LABEL              CODE        MNEMONIC          PAGE#: 3

7F    56                          7F
7F    57                          0E          MVI   C,2
7F    58                          02
7F    59                          CA          JZ    LOADB
7F    5A                          1A  @
7F    5B                          7F
7F    5C                          04          INR   B
7F    5D                          78          MOV   AB
7F    5E                          D3          OUT   .193
7F    5F                          C1
7F    60                          3E          MVI   A,.17
7F    61                          11
7F    62                          CD          CALL  LOOP
7F    63                          6F  @
7F    64                          7F
7F    65                          3E          MVI   A,.9
7F    66                          09
7F    67                          CD          CALL  LOOP
7F    68                          6F  @
7F    69                          7F
7F    6A                          0E          MVI   C,1
7F    6B                          01
7F    6C                          C3          JMP   LOADB
7F    6D                          1A  @
7F    6E                          7F

*-DELAY LOOP SUBROUTINE-*

7F    6F        <LOOP>            D3          OUT   .192
7F    70                          C0
7F    71        <LOOPA>           97          SUB   A
7F    72                          D3          OUT   .192
7F    73                          C0
7F    74        <LOOPB>           DB          IN    .192
7F    75                          C0
7F    76                          1F          RAR
7F    77                          DA          JC    LOOPB
7F    78                          74  @
7F    79                          7F
7F    7A                          C9          RET

*-OPUS LOADED: OPTIONALLY INITIALIZE PORTS-*

*-THE FOLLOWING INITIALIZATION FOR IMSAI SIO LOW PORT 32-*

7F    7B        <END>             3E          MVI   A,.129
7F    7C                          81
7F    7D                          D3          OUT   .35
7F    7E                          23
7F    7F                          3E          MVI   A,.64
7F    80                          40
7F    81                          D3          OUT   .35
7F    82                          23
7F    83                          3E          MVI   A,.74
```

PAGE	LOC	LABEL	CODE	MNEMONIC	
7F	84		4A		
7F	85		D3	OUT	,35
7F	86		23		
7F	87		3E	MVI	A,,39
7F	88		27		
7F	89		D3	OUT	,35
7F	8A		23		
7F	8B		AF	XRA	A
7F	8C		D3	OUT	,34
7F	8D		22		
7F	8E		C3	JMP	0
7F	8F		00		
7F	90		00		

Paper Tape Loader

Paper tapes of OPUS with the System Generation Routine will be received in Intel Hex format. This loader assumes that format. The byte input routine in the loader is for MITS SIOC teletype or Processor Technology 3P+S (serial) interface boards. The port numbers are Ø and 1. This input routine may be easily modified to reflect a different interface.

In Intel Hex format, OPUS is divided into blocks. Except for the first character of a block, all bytes are divided into two Hex characters. The high-order four bits and the low-order four bits are the two characters. Each character is converted to an ASCII character, Ø - 9 or A - F. If the initial four bits represents a number less than 10, then 48 is added to produce ASCII Ø through 9; otherwise 55 is added to produce ASCII A through F. The high-order character precedes the low-order character sequentially.

In each block, there are 9 header characters defining the following:

Character #	Description
1	":" Block initiator
2,3	Number of bytes in the block; this will be Ø when no more data is present
4,5	High-order starting location in memory
6,7	Low-order starting location in memory
8,9	Not used in OPUS dumps

Following the header bytes is the number of specified data bytes of OPUS, each byte represented by two Hex characters.

The last byte of the block (2 characters) is the checksum byte. This byte will be the complement of the sum of all data bytes.

The loader sequentially reads the data, putting each data byte into the correct location in memory. If a checksum error is detected, the computer will halt. After all data is in, execution will be immediately sent to the start of OPUS.

NOTE: It is important to note that only the OPUS with the System Generation Routine will be in this format. If initialized OPUS is dumped on a paper tape, the format will be identical to that listed under the cassette section. It will be in straight Binary form.

```
PAGE  LOC      LABEL            CODE       MNEMONIC          PAGE#: 1

*****  LOADER FOR INTEL HEX FORMATTED PAPER TAPE  *****

177   000      <LOAD>           061        LXI    S,LOAD
177   001                       000 @
177   002                       177

*-WAIT FOR START OF BLOCK-*

177   003      <LOADA>          315        CALL   INP
177   004                       131 @
177   005                       177
177   006                       376        CPI    ":"
177   007                       072
177   010                       302        JNZ    LOADA
177   011                       003 @
177   012                       177

*-GET BLOCK SIZE & CHECK FOR END-OF-FILE-*

177   013                       315        CALL   DINP
177   014                       072 @
177   015                       177
177   016                       312        JZ     END
177   017                       061 @
177   020                       177

*-GET REST OF BLOCK PARAMETERS-*

177   021                       127        MOV    DA
177   022                       315        CALL   DINP
177   023                       072 @
177   024                       177
177   025                       147        MOV    HA
177   026                       315        CALL   DINP
177   027                       072 @
177   030                       177
177   031                       157        MOV    LA
177   032                       315        CALL   DINP
177   033                       072 @
177   034                       177
177   035                       006        MVI    B,0
177   036                       000

*-READ IN DATA-*

177   037      <LOADB>          315        CALL   DINP
177   040                       072 @
177   041                       177
177   042                       167        MOV    MA
177   043                       043        INX    H
177   044                       025        DCR    D
```

```
PAGE  LOC          LABEL          CODE        MNEMONIC          PAGE#: 2

177   045                         302         JNZ   LOADB
177   046                         037 @
177   047                         177
177   050                         110         MOV   CB

*-GET AND COMPARE CHECKSUM-*

177   051                         315         CALL  DINP
177   052                         072 @
177   053                         177
177   054                         201         ADD   C
177   055                         312         JZ    LOADA
177   056                         003 @
177   057                         177

*-CHECKSUM ERROR: HALT-*

177   060                         166         HLT

*-LOAD COMPLETE: START EXECUTION-*

177   061     <END>               315         CALL  DINP
177   062                         072 @
177   063                         177
177   064                         147         MOV   HA
177   065                         315         CALL  DINP
177   066                         072 @
177   067                         177
177   070                         157         MOV   LA
177   071                         351         PCHL

*****  SUBROUTINES  *****


*-INPUT DOUBLE BYTE, CONVERT TO BINARY-*

177   072     <DINP>              315         CALL  INP
177   073                         131 @
177   074                         177
177   075                         315         CALL  HEX
177   076                         115 @
177   077                         177
177   100                         007         RLC
177   101                         007         RLC
177   102                         007         RLC
177   103                         007         RLC
177   104                         137         MOV   EA
177   105                         315         CALL  INP
177   106                         131 @
177   107                         177
177   110                         315         CALL  HEX
177   111                         115 @
177   112                         177
177   113                         203         ADD   E
177   114                         311         RET
```

PAGE	LOC	LABEL	CODE	MNEMONIC	
		-CONVERT HEX CHARACTER TO BINARY-			
177	115	<HEX>	365	PUSH	S
177	116		200	ADD	B
177	117		107	MOV	BA
177	120		361	POP	S
177	121		326	SUI	.48
177	122		060		
177	123		376	CPI	.10
177	124		012		
177	125		330	RC	
177	126		326	SUI	.7
177	127		007		
177	130		311	RET	
		-INPUT BYTE, LOW PORT 0-			
177	131	<INP>	333	IN	0
177	132		000		
177	133		017	RRC	
177	134		332	JC	INP
177	135		131 @		
177	136		177		
177	137		333	IN	1
177	140		001		
177	141		346	ANI	.127
177	142		177		
177	143		311	RET	

```
PAGE  LOC       LABEL           CODE      MNEMONIC          PAGE#: 1

*****  LOADER FOR INTEL HEX FORMATTED PAPER TAPE  *****

7F    00        <LOAD>          31        LXI    S,LOAD
7F    01                        00  @
7F    02                        7F

*-WAIT FOR START OF BLOCK-*

7F    03        <LOADA>         CD        CALL   INP
7F    04                        59  @
7F    05                        7F
7F    06                        FE        CPI    ":"
7F    07                        3A
7F    08                        C2        JNZ    LOADA
7F    09                        03  @
7F    0A                        7F

*-GET BLOCK SIZE & CHECK FOR END-OF-FILE-*

7F    0B                        CD        CALL   DINP
7F    0C                        3A  @
7F    0D                        7F
7F    0E                        CA        JZ     END
7F    0F                        31  @
7F    10                        7F

*-GET REST OF BLOCK PARAMETERS-*

7F    11                        57        MOV    DA
7F    12                        CD        CALL   DINP
7F    13                        3A  @
7F    14                        7F
7F    15                        67        MOV    HA
7F    16                        CD        CALL   DINP
7F    17                        3A  @
7F    18                        7F
7F    19                        6F        MOV    LA
7F    1A                        CD        CALL   DINP
7F    1B                        3A  @
7F    1C                        7F
7F    1D                        06        MVI    B,0
7F    1E                        00

*-READ IN DATA-*

7F    1F        <LOADB>         CD        CALL   DINP
7F    20                        3A  @
7F    21                        7F
7F    22                        77        MOV    MA
7F    23                        23        INX    H
7F    24                        15        DCR    D
```

```
PAGE  LOC        LABEL          CODE        MNEMONIC
----  ---  ------------------  ----  ----------------------

7F    25                       C2        JNZ   LOADB
7F    26                       1F  @
7F    27                       7F
7F    28                       48        MOV   CB

*-GET AND COMPARE CHECKSUM-*

7F    29                       CD        CALL  DINP
7F    2A                       3A  @
7F    2B                       7F
7F    2C                       81        ADD   C
7F    2D                       CA        JZ    LOADA
7F    2E                       03  @
7F    2F                       7F

*-CHECKSUM ERROR: HALT-*

7F    30                       76        HLT

*-LOAD COMPLETE: START EXECUTION-*

7F    31       <END>           CD        CALL  DINP
7F    32                       3A  @
7F    33                       7F
7F    34                       67        MOV   HA
7F    35                       CD        CALL  DINP
7F    36                       3A  @
7F    37                       7F
7F    38                       6F        MOV   LA
7F    39                       E9        PCHL

*****  SUBROUTINES  *****

*-INPUT DOUBLE BYTE, CONVERT TO BINARY-*

7F    3A       <DINP>          CD        CALL  INP
7F    3B                       59  @
7F    3C                       7F
7F    3D                       CD        CALL  HEX
7F    3E                       4D  @
7F    3F                       7F
7F    40                       07        RLC
7F    41                       07        RLC
7F    42                       07        RLC
7F    43                       07        RLC
7F    44                       5F        MOV   EA
7F    45                       CD        CALL  INP
7F    46                       59  @
7F    47                       7F
7F    48                       CD        CALL  HEX
7F    49                       4D  @
7F    4A                       7F
7F    4B                       83        ADD   E
7F    4C                       C9        RET
```

PAGE	LOC	LABEL	CODE	MNEMONIC	
		-CONVERT HEX CHARACTER TO BINARY-			
7F	4D	<HEX>	F5	PUSH	S
7F	4E		80	ADD	B
7F	4F		47	MOV	BA
7F	50		F1	POP	S
7F	51		D6	SUI	.48
7F	52		30		
7F	53		FE	CPI	.10
7F	54		0A		
7F	55		D8	RC	
7F	56		D6	SUI	.7
7F	57		07		
7F	58		C9	RET	
		-INPUT BYTE, LOW PORT 0-			
7F	59	<INP>	DB	IN	0
7F	5A		00		
7F	5B		0F	RRC	
7F	5C		DA	JC	INP
7F	5D		59 @		
7F	5E		7F		
7F	5F		DB	IN	1
7F	60		01		
7F	61		E6	ANI	.127
7F	62		7F		
7F	63		C9	RET	

Cassette Loader

OPUS dumps on cassettes will be for MITS ACR interfaces. Ports 6 and 7 are assumed. These may be easily changed in the loader.

OPUS is dumped on cassettes in blocks of data. Each block consists of 137 Binary bytes. The first byte is a checksum byte: the sum of all data bytes in the block with bits 6 and 7 forced high (ORed with 3ØØ Octal). This checksum byte will be a zero when there is no more data to load. The remaining 136 data bytes are Binary OPUS code.

There will be a leader of at least 1ØØ null bytes (Ø) at the beginning and at the end of the tape.

The first non-zero byte on the tape will be a 255 to designate the start of the file. The first block immediately follows. Blocks are dumped sequentially, i.e., the first block contains the first 136 bytes of OPUS, the second block contains the next 136 bytes, etc.

After loading all data, this loader will initialize a port for the MITS 2SIO board and immediately begin execution of OPUS. This may be modified.

NOTE: When dumping initialized OPUS in the System Generation Routine, if a serial device is specified, the above format describes the manner in which OPUS will be dumped. This includes paper tapes, terminals, cassettes, etc.

Cassette Loader: Octal

```
PAGE  LOC      LABEL          CODE       MNEMONIC          PAGE#: 1

*****  LOADER FOR MITS CASSETTE  *****

177   000      <LOAD>         333        IN     .7
177   001                     007
177   002                     061        LXI    S,LOAD
177   003                     000 @
177   004                     177
177   005                     041        LXI    H,0
177   006                     000
177   007                     000

*-WAIT FOR START-*

177   010      <LOADA>        315        CALL   INP
177   011                     054 @
177   012                     177
177   013                     267        ORA    A
177   014                     312        JZ     LOADA
177   015                     010 @
177   016                     177

*-READ CHECKSUM, CHECK FOR END-*

177   017      <LOADB>        315        CALL   INP
177   020                     054 @
177   021                     177
177   022                     267        ORA    A
177   023                     312        JZ     END
177   024                     065 @
177   025                     177
177   026                     107        MOV    BA
177   027                     021        LXI    D,.136
177   030                     210
177   031                     000

*-READ BLOCK OF 136 BYTES-*

177   032      <LOADC>        315        CALL   INP
177   033                     054 @
177   034                     177
177   035                     167        MOV    MA
177   036                     043        INX    H
177   037                     202        ADD    D
177   040                     127        MOV    DA
177   041                     035        DCR    E
177   042                     302        JNZ    LOADC
177   043                     032 @
177   044                     177

*-COMPARE CHECKSUM-*
```

```
PAGE  LOC        LABEL            CODE       MNEMONIC
----  ---  ---------------------  ----  --------------------

177   045                         366     ORI   .192
177   046                         300
177   047                         270     CMP   B
177   050                         312     JZ    LOADB
177   051                         017 @
177   052                         177

*-CHECKSUM ERROR-*

177   053                         166     HLT

*-SUBROUTINE TO INPUT BYTE-*

177   054      <INP>              333     IN    .6
177   055                         006
177   056                         017     RRC
177   057                         332     JC    INP
177   060                         054 @
177   061                         177
177   062                         333     IN    .7
177   063                         007
177   064                         311     RET

*-LOAD COMPLETE: INITIALIZE PORT IF NECESSARY-*


*-MITS 2SIO PORT INITIALIZED-*

177   065      <END>              076     MVI   A,.3
177   066                         003
177   067                         323     OUT   .16
177   070                         020
177   071                         076     MVI   A,.21
177   072                         025
177   073                         323     OUT   .16
177   074                         020
177   075                         303     JMP   0
177   076                         000
177   077                         000
```

Cassette Loader: Hex

```
PAGE  LOC       LABEL              CODE       MNEMONIC          PAGE#: 1

*****  LOADER FOR MITS CASSETTE  *****

7F    00        <LOAD>             DB         IN     .7
7F    01                           07
7F    02                           31         LXI    S,LOAD
7F    03                           00   @
7F    04                           7F
7F    05                           21         LXI    H,0
7F    06                           00
7F    07                           00

*-WAIT FOR START-*

7F    08        <LOADA>            CD         CALL   INP
7F    09                           2C   @
7F    0A                           7F
7F    0B                           B7         ORA    A
7F    0C                           CA         JZ     LOADA
7F    0D                           08   @
7F    0E                           7F

*-READ CHECKSUM, CHECK FOR END-*

7F    0F        <LOADB>            CD         CALL   INP
7F    10                           2C   @
7F    11                           7F
7F    12                           B7         ORA    A
7F    13                           CA         JZ     END
7F    14                           35   @
7F    15                           7F
7F    16                           47         MOV    BA
7F    17                           11         LXI    D,.136
7F    18                           88
7F    19                           00

*-READ BLOCK OF 136 BYTES-*

7F    1A        <LOADC>            CD         CALL   INP
7F    1B                           2C   @
7F    1C                           7F
7F    1D                           77         MOV    MA
7F    1E                           23         INX    H
7F    1F                           82         ADD    D
7F    20                           57         MOV    DA
7F    21                           1D         DCR    E
7F    22                           C2         JNZ    LOADC
7F    23                           1A   @
7F    24                           7F

*-COMPARE CHECKSUM-*
```

```
PAGE  LOC        LABEL          CODE        MNEMONIC
----  ---  ----------------     ----  ------------------

7F    25                        F6        ORI    .192
7F    26                        C0
7F    27                        B8        CMP    B
7F    28                        CA        JZ     LOADB
7F    29                        0F  @
7F    2A                        7F

*-CHECKSUM ERROR-*

7F    2B                        76        HLT

*-SUBROUTINE TO INPUT BYTE-*

7F    2C       <INP>            DB        IN     .6
7F    2D                        06
7F    2E                        0F        RRC
7F    2F                        DA        JC     INP
7F    30                        2C  @
7F    31                        7F
7F    32                        DB        IN     .7
7F    33                        07
7F    34                        C9        RET

*-LOAD COMPLETE: INITIALIZE PORT IF NECESSARY-*


*-MITS 2SIO PORT INITIALIZED-*

7F    35       <END>            3E        MVI    A,.3
7F    36                        03
7F    37                        D3        OUT    .16
7F    38                        10
7F    39                        3E        MVI    A,.21
7F    3A                        15
7F    3B                        D3        OUT    .16
7F    3C                        10
7F    3D                        C3        JMP    0
7F    3E                        00
7F    3F                        00
```

C. DISC & FILE FORMAT

The format of discs operating under all A.S.I. software will vary according to the particular drive being implemented.

The following parameter values are stored in a table in all operating system software, defining the format of the particular disc:

Value	Description	Formula #
NB	Number of data bytes per sector	
NS	Number of sectors per track	
NT	Number of tracks per disc	
SP	Number of sectors in disc parameter table	(1)
SD	Number of sectors in directory	(2)
SS	Number of data sectors per disc	(3)
BP	Size of disc parameter table	(4)
BF	Size of sector-free table	(5)
BD	Size of directory-free table	(6)
DB	Number of bytes per directory logical record	(7)
DR	Number of directory records per sector	(8)
DD	Number of directory records	(9)

Formulas for Value Calculation	Formula #
1. SS = (NT -1) * NS	(3)
BF = INT ((SS - 1) / 8) + 1	(5)
DB = Minimum 17	(7)
DR = INT (NB / DB)	(8)
SP = INT ((BF - 1) / NB) + 1	(1)
2. X = SP * NB	
SD = NS - SP	(2)
DD = SD * DR	(9)
BD = INT ((DD - 1) / 8) + 1	(6)
BP = BF + BD + 8	(4)
3. IF BP > X: SP = SP + 1; GO TO STEP 2.	

Table Descriptions

A. Disc Parameter Table

Three buffers constitute the disc parameter table, which resides both on disc and in memory during operating system execution.

They sequentially follow each other on the disc, overlapping sector boun-

daries, in the following order:

1. Sector-Free Table

   The sector-free table keeps the status of all data sectors on the disc, enabling the operating system to find free sectors for data and to return full sectors to free status.

   Each data sector on the disc requires one bit in this table with the value:

   Ø: Sector empty (available for data)
   1: Sector full (in use by some file)

   The location of any data sector (Track T, Sector S) in this table may be calculated as:

```
POS = (T -1) * NS + S
BYTE = INT (POS / 8)
BIT = POS - BYTE * 8
```

2. Directory-Free Table

   The directory-free table keeps the status of all directory records, enabling the operating system to find a free record quickly for a new file name or to return a record to free status.

   Each directory logical record (see definition below) requires one bit in this table with the value:

   Ø: Directory record empty
   1: Directory record full

   The location of a directory record (Sector S, Logical Record L) in this table may be calculated as:

```
POS = (S - SP) * DR + L
BYTE = INT (POS / 8)
BIT = POS - BYTE * 8
```

3. Disc Tag

   The disc tag (identification label of any ASCII character string not exceeding a length of 7 bytes) follows the directory-free table.

   The disc tag buffer requires 8 bytes.

B. Directory

Every program or file stored on the disc requires one directory record in the directory to store such data as the name, file type, and location of the file. The parameters in the record depend solely on the type of file.

Each record requires a minimum of 17 bytes. Depending upon the value of NB, this number may be maximized to distribute wasted bytes throughout the records in a sector, possibly allowing room for additional file parameters.

The following tables describe the lay-out of the directory record:

Byte	Description
0-6	File name (ASCII character string)
7	File type (ASCII character; see below)
8-16	File parameters (see below)

Type	Description	Byte	Description
$	ASCII sequential data file	8	Starting sector
		9	Starting track
		10-11	Number of bytes in file
		12-13	Number of sectors in file
		14-16	Not used
B	ASCII back-up data file	8	Starting sector
		9	Starting track
		10-11	Number of bytes in file
		12-13	Number of sectors in file
		14-16	Not used
R	Relocatable assembly program	8	Starting sector: machine code
		9	Starting track: machine code
		10-11	Number of bytes: machine code
		12	Starting sector: label table
		13	Starting track: label table
		14-15	Number of bytes: label table
		16	Not used
@	Batch ASCII output file	8	Starting sector
		9	Starting track
		10-11	Number of bytes
		12-13	Number of sectors
		14-16	Not used
S	OPUS source program	8	Starting sector
		9	Starting track
		10-11	Number of bytes
		12-13	Number of sectors
		14-16	Not used
O	OPUS object program	8	Starting sector
		9	Starting track
		10-11	Number of bytes
		12-13	Number of sectors
		14-16	Not used

Type	Description	Byte	Description
F	OPUS random data file	8	Starting sector: data map
		9	Starting track: data map
		10-11	Maximum # of logical records
		12-15	Not used
		16	# of logical records per sector
D	OPUS random dimensioned data file	8	Starting sector: data map
		9	Starting track: data map
		10	Max. value of highest dim.
		11	
		12	.
		13	.
		14	.
		15	Max. value of lowest dim.
		16	# of logical records per sector

If other file types are specified, they will normally be sequential files with the same parameters as type $.

C. Data Sectors

The rest of the disc is available for data, including all programs and data files. These may be broken down into two file types:

1. Sequential Files

Sequential files must be accessed sequentially through each data sector. Data sectors may reside anywhere on the disc and are linked by means of a pointer.

The first two bytes of all sequential files are reserved and indicate the following:

Byte Ø: Sector number of next record in the file
Byte 1: Track number of next record in the file

The last sector of the file has both bytes = Ø. The remaining bytes of a sector are available for data.

2. Random Files

Any data logical record in a random file may be accessed with a maximum of two disc reads.

Random files are broken down into two sections:

a. Data Map

The data map contains pointers to all data sectors in the file.

The sectors of the map are contiguous, the size being determined by both the number of logical records per sector (LR) and the maximum number of logical records (MAX) in the file.

```
SIZE = INT ((INT ((MAX - 1) / LR)) / NB / 2) - 1
```

Each sector of the map contains (NB/2) number of logical records, each logical record consisting of 2 bytes:

```
Byte Ø:  Sector number of data sector
Byte 1:  Track number of data sector
```

Given a file logical record, L, the data sector may be found in the map:

```
DS = INT ((L - 1) / LR)      Data sector position
DL = L - DS * LR             LR in data sector
MS = INT (DS / NB / 2)       Map sector position
ML = DS - MS * NB / 2        LR in map sector
```

b. Data Sector

Each sector of data may contain 1 or more logical records, depending upon user specifications. Each logical record contains any data, and is terminated by either a null byte (Ø) should the data require less than the reserved number of bytes, or a software-determined boundary should the data require the exact number of reserved bytes.

Data sectors are not contiguous and may reside anywhere on the disc in any order.

Disc Lay-Out

Track	Sector	Description
Ø	Ø through SP-1	Disc parameter table
Ø	SP through NS-1	Directory
1 through NT-1	Ø through NS-1	Data sectors

D. STATEMENT TABLE

The table on the following pages gives a brief description and the characteristics of every statement available in OPUS. These definitions describe the table categories:

NO: Statement Number. This number is returned when a statement is read during a SCAN statement.

STMNT: Statement. Statements must be used exactly as they are shown in this column.

NAME: Statement Name Unabbreviated.

DESCRIPTION: Description of the statement operation.

STMNT TYPE: Type of operation:

C: Command	O: Operator
F: Function	S: Statement (unclassified)

EXEC MODE: Execution mode in which the statement may be implemented:

- C: During command mode only
- P: In a program only
- E: Either during command mode or in a program

ARG FORMAT: Argument format:

- #: The argument(s) will be converted to a number prior to operation
- $: The argument(s) will be converted to a string prior to operation
- A: Any type of argument (number, string or matrix) allowed
- E: Either a number or string allowed as the argument value
- M: The argument must be a matrix variable
- N: No arguments are used
- O: Ordered mixture of argument formats, structured according to the operation

ARG REQ: Number of arguments required by the statement:

- n : The statement uses exactly this number of arguments
- n-n: The first is the minimum number and the second the maximum number of arguments
- n- : This is the minimum number; there is no maximum number of arguments (the statement uses all operands in the operand table)

ARG RET: Number of arguments returned to the operand table by the operation

PRIOR: The priority value determining the order of execution. Larger numbers connote a higher priority than smaller numbers and will be executed first should there be a mixture of statements in an expression.

Asterisks (*) appearing in any of the categories indicate that the statement is non-executable, i.e., it does not exist in the object code, but only in source code.

NO	STMNT	NAME	DESCRIPTION	STMNT TYPE	EXEC MODE	ARG FORMAT	# ARG REQ	# ARG RET	PRIOR
0	ASSIGN	ASSIGN	ASSIGN A FILE NUMBER TO A DISC FILE	C	E	0	2-3	0	2
1	BRK	BReaK	ENABLE/DISABLE INTERRUPTS FROM THE INPUT DEVICE	C	E	#	1	0	2
2	COM	COMpile	COMPILE SOURCE CODE TO OBJECT CODE	C	C	$	0-1	0	2
3	CONT	CONTinue	DETERMINE THE END OF A WHILE...CONT LOOP	C	E	N	0	0	2
4	CSAVE	Compiled SAVE	DUMP AN OBJECT PROGRAM TO A PERIPHERAL DEVICE	C	E	0	1-2	0	2
5	DEL	DELete	DELETE A SPECIFIED SECTION OF A SOURCE PROGRAM	C	E	#	1-2	0	2
6	DIM	DIMension	DETERMINE MATRIX VARIABLE; ALLOCATE BUFFER	C	E	M	1	0	2
7	EFILE	End of FILE	DECLARE IF END-OF-FILE IS TO BE PROGRAM CONTROLLED	C	E	#	2	0	2
8	ELSE	ELSE	EXECUTE A BLOCK OF CODE IF A FALSE CONDITION EXISTS	C	E	N	0	0	2
9	END	END	TERMINATE PROGRAM EXECUTION	C	E	N	0	0	2
10	GET	GET	LOAD A SOURCE PROGRAM FROM A PERIPHERAL DEVICE	C	E	0	1-2	0	2
11	GOSUB	GO SUBroutine	CALL A SUBROUTINE IN THE PROGRAM	C	E	$	1	0	2
12	GOTO	GO TO	EXECUTE AN UNCONDITIONAL JUMP TO A PROGRAM LOCATION	C	E	$	1	0	2
13	IF	IF	EXECUTE A BLOCK OF CODE IF A TRUE CONDITION EXISTS	C	E	#	1	0	2
14	IN	INput	DECLARE THE INPUT DEVICE	C	E	#	1	0	2
15	INPUT	INPUT	REQUEST DATA INPUT FROM A PERIPHERAL DEVICE	C	E	0	1-	0	2
16	KILL	KILL	PURGE A PROGRAM OR FILE FROM THE DISC	C	E	0	1-2	0	2
17	LIB	LIBrary	LIST DISC LIBRARY OF FILES AND PROGRAMS	C	E	#	0-2	0	2
18	LIN	LINe	GENERATE LINE FEEDS TO THE OUTPUT DEVICE	C	E	#	1-2	0	2
19	LIST	LIST	LIST A SOURCE PROGRAM TO AN OUTPUT DEVICE	C	E	#	0-5	0	2
20	LOAD	LOAD	LOAD AN OBJECT PROGRAM FROM A PERIPHERAL DEVICE	C	E	0	1-2	0	2
21	LOOP	LOOP	EXECUTE A SECTION OF CODE A SPECIFIED NUMBER OF TIMES	C	E	0	2	0	2
22	NEW	NEW	CLEAR PROGRAM AREA	C	C	$	0-1	0	2
23	NEXT	NEXT	DECLARE THE END OF A LOOP OPERATION	C	E	#	0-1	0	2
24	ON	ON	EXECUTE A SPECIFIC BLOCK OF CODE	C	E	#	1	0	2
25	OPEN	OPEN	CREATE A DATA FILE ON THE DISC	C	E	0	3-4	0	2

NO	STMNT	NAME	DESCRIPTION	STMNT TYPE	EXEC MODE	ARG FORMAT	# ARG REQ	# ARG RET	PRIOR
26	OUT	OUTput	DECLARE THE OUTPUT DEVICE	C	E	#	1	0	2
27	PF	Print Formatted	GENERATE DATA TO AN OUTPUT DEVICE IN FORMATTED FIELDS	C	E	0	1-	0	2
28	PRINT	PRINT	GENERATE DATA TO AN OUTPUT DEVICE	C	E	0	0-	0	2
29	PURGE	PURGE	DELETE ALL DATA IN A DISC FILE LOGICAL RECORD	C	E	#	2	0	2
30	READ	READ	READ DATA FROM A DISC FILE	C	E	0	3-	0	2
31	REN	RENumber	RENUMBER THE LINES OF A SOURCE PROGRAM	C	E	#	0-4	0	2
32	RETURN	RETURN	EXIT FROM A SUBROUTINE TO THE MAIN PROGRAM	C	E	N	0	0	2
33	RUN	RUN	START EXECUTION OF AN OBJECT PROGRAM	C	C	N	0	0	2
34	SAVE	SAVE	DUMP A SOURCE PROGRAM TO A PERIPHERAL DEVICE	C	E	0	1-2	0	2
35	SCAN	SCAN	READ DATA CONTAINED IN THE PROGRAM	C	E	0	1-	0	2
36	SPA	SPAce	GENERATE SPACES TO THE OUTPUT DEVICE	C	E	#	1	0	2
37	STUFF	STUFF	WRITE A BYTE INTO A MEMORY LOCATION	C	E	#	2	0	2
38	THEN	THEN	SPECIFY ARGUMENTS FOR ANOTHER OPERATION	C	E	E	0-	0-	2
39	TO	TO	DETERMINE MAXIMUM NUMBER OF LOOPS IN A LOOP OPERATION	C	E	#	1	0	2
40	WHILE	WHILE	EXECUTE A SECTION OF CODE WHILE A CONDITION IS TRUE	C	E	#	1	0	2
41	WRITE	WRITE	WRITE DATA ON A DISC FILE	C	E	0	3-	0	2
42	ABS	ABSolute	DETERMINE THE ABSOLUTE VALUE OF THE ARGUMENT	F	E	#	1	1	12
43	ASC	ASCii	DETERMINE THE ASCII CHARACTER OF THE ARGUMENT	F	E	#	1	1	12
44	ATN	ArcTaNgent	DETERMINE THE ARCTANGENT OF THE ARGUMENT	F	E	#	1	1	12
45	COS	COSine	DETERMINE THE COSINE OF AN ANGLE	F	E	#	1	1	12
46	DATE	DATE	DETERMINE THE CURRENT DAY, MONTH, OR YEAR	F	E	#	1	1	12
47	EOF	End Of File	DETERMINE IF AN END-OF-FILE HAS BEEN REACHED	F	E	#	1	1	12
48	EXP	EXPonent	DETERMINE EXPONENTIAL E RAISED TO A POWER	F	E	#	1	1	12
49	FETCH	FETCH	RETRIEVE THE CONTENTS OF A MEMORY LOCATION	F	E	#	1	1	12
50	FILE	FILE	DETERMINE SPECIFICATIONS ABOUT A DISC FILE	F	E	#	2	1	12

NO	STMNT	NAME	DESCRIPTION	STMNT TYPE	EXEC MODE	ARG FORMAT	# ARG REQ	# ARG RET	PRIOR
51	LEN	LENgth	DETERMINE THE NUMBER OF CHARACTERS IN THE ARGUMENT	F	E	$	1	1	12
52	LOG	LOGarithm	DETERMINE THE NATURAL LOGARITHM OF THE ARGUMENT	F	E	#	1	1	12
53	NONE	NONE	DETERMINE IF DATA ENTERED DURING PRIOR INPUT	F	E	N	0	1	13
54	NOT	NOT	LOGICALLY NEGATE THE ARGUMENT	F	E	#	1	1	12
55	NUM	NUMber	DETERMINE THE NUMBER FORMAT OF THE ARGUMENT	F	E	#	1	1	12
56	RND	RaNDom	DETERMINE A RANDOM NUMBER BETWEEN 0 AND 1	F	E	N	0	1	12
57	SGN	SiGN	DETERMINE THE SIGN OF A NUMBER	F	E	#	1	1	12
58	SIN	SINe	DETERMINE THE SINE OF AN ANGLE	F	E	#	1	1	12
59	SQR	SQuare Root	DETERMINE THE SQUARE ROOT OF A NUMBER	F	E	#	1	1	12
60	STR	STRing	DETERMINE THE STRING FORMAT OF THE ARGUMENT	F	E	$	1	1	12
61	TAN	TANgent	DETERMINE THE TANGENT OF AN ANGLE	F	E	#	1	1	12
62	TRU	TRUncate	TRUNCATE A NUMBER TO AN INTEGER	F	E	#	1	1	12
63	#	#	DETERMINE WHETHER OR NOT TWO EXPRESSIONS ARE EQUAL	0	E	E	2	1	8
64	&	&	CONCATENATE TWO STRINGS	0	E	$	2	1	9
65	*	*	MULTIPLY TWO NUMBERS	0	E	#	2	1	10
66	+	+	ADD TWO NUMBERS	0	E	#	2	1	9
67	-	-	NEGATE THE ARGUMENT	0	E	#	1	1	12
68	-	-	SUBTRACT TWO NUMBERS	0	E	#	2	1	9
69	/	/	DIVIDE TWO NUMBERS	0	E	#	2	1	10
70	<=	<=	DETERMINE IF ONE EXPRESSION IS LESS THAN OR EQUAL TO ANOTHER	0	E	E	2	1	8
71	=	=	ASSIGN A VALUE TO A VARIABLE	0	E	0	2	1	5
72	>=	>=	DETERMINE IF ONE EXPRESSION IS GREATER THAN OR EQUAL TO ANOTHER	0	E	E	2	1	8

NO	STMNT	NAME	DESCRIPTION	STMNT TYPE	EXEC MODE	ARG FORMAT	# ARG REQ	# ARG RET	PRIOR
73	AND	AND	DETERMINE IF TWO EXPRESSIONS ARE BOTH TRUE	0	E	#	2	1	7
74	IS	IS	DETERMINE IF ONE EXPRESSION IS THE SAME AS ANOTHER	0	E	E	2	1	8
75	MAX	MAXimum	DETERMINE WHICH EXPRESSION HAS THE GREATER VALUE	0	E	#	2	1	8
76	MIN	MINimum	DETERMINE WHICH EXPRESSION HAS THE LESSER VALUE	0	E	#	2	1	8
77	OR	OR	DETERMINE IF ONE EXPRESSION HAS A TRUE VALUE	0	E	#	2	1	6
78	↑	↑	RAISE ONE NUMBER TO THE POWER OF ANOTHER	0	E	#	2	1	11
79	(	(	SUBSTRING/MATRIX LEFT DELIMITER	S	*	*	*	*	0
80	)	)	SUBSTRING/MATRIX RIGHT DELIMITER	S	*	*	*	*	1
81	,	,	LIST OR OPERATION DELIMITER	S	*	*	*	*	4
82	:	:	LIST DELIMITER WITHIN AN OPERATION	S	E	E	1-2	0	4
83	;	;	STATEMENT DELIMITER; CLEARS OPERAND TABLE	S	E	A	0-	0	1
84	REM	REMark	ALLOW ENTRY OF A COMMENT IN THE SOURCE PROGRAM	S	*	*	*	*	2
85	[	[	LEFT HAND DELIMITER OF A BLOCK OF CODE	S	E	N	0	0	1
86	\	\	DELIMIT BLOCKS OF CODE	S	E	N	0	0	4
87	]	]	RIGHT HAND DELIMITER OF A BLOCK OF CODE	S	E	N	0	0	1
88	!	!	MATRIX ELEMENT REFERENCE	F	E	0	2-	1	15
89	$	$	SUBSTRING REFERENCE	F	E	0	2-3	1	15
90									
91	<	<	DETERMINE IF ONE EXPRESSION IS LESS THAN ANOTHER	0	E	E	2	1	8
92	>	>	DETERMINE IF ONE EXPRESSION IS GREATER THAN ANOTHER	0	E	E	2	1	8
93	SET	SET	DECLARE BUFFER SIZES & SET PRECISION	C	E	#	2	0	2
94	BYE	BYE	TERMINATE OPUS/ONE	C	E	N	0	0	2
95	CLOSE	CLOSE	CLOSE A DISC FILE	C	E	#	0-1	0	2
96	DISC	DISC	DECLARE AND ENABLE A DISC DRIVE	C	E	E	1	0	2

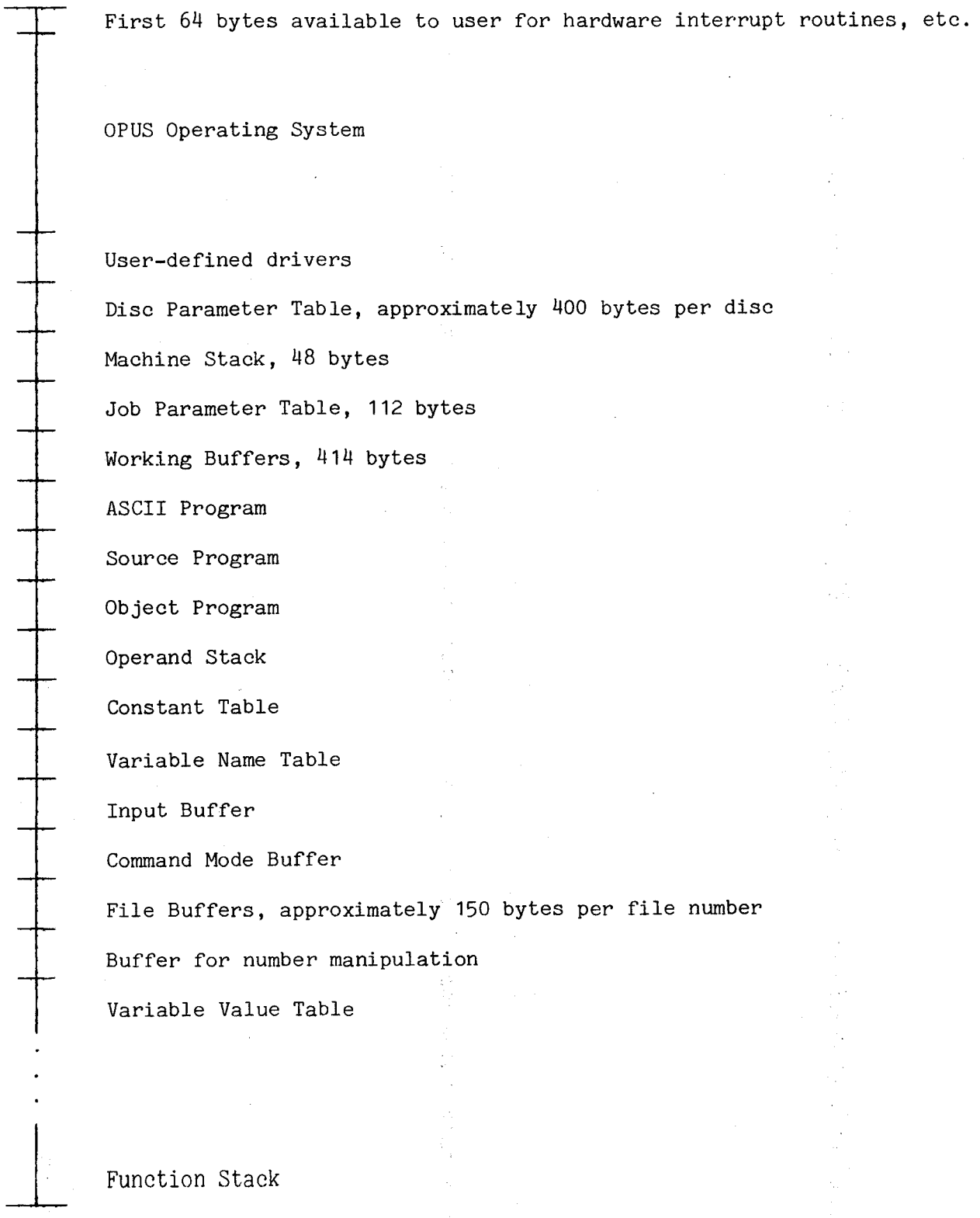
NO	STMNT	NAME	DESCRIPTION	STMNT TYPE	EXEC MODE	ARG FORMAT	# ARG REQ	# ARG RET	PRIOR
97	ERR	ERRor	DECLARE ERROR SUBROUTINE	C	E	$	1	0	2
98	DATA	DATA	FILL MATRIX IMMEDIATELY WITH DATA	C	E	M	2-	0	2
99	GLOBAL	GLOBAL	DECLARE GLOBAL VARIABLES	C	E	0	2-	0	2
100	CALL	CALL	CALL AN EXTERNAL SUBROUTINE	C	E	0	1-	0	13
101	@	@	CALL AN EXTERNAL FUNCTION	F	E	0	1-	0-	13
102	EXT	EXTernal	DECLARE AN EXTERNAL SUBROUTINE	C	E	0	1-2	0	2
103	SUB	SUBroutine	SPECIFY START OF AN EXTERNAL SUBROUTINE	C	P	0	2-	0	2
104	RET	RETurn	RETURN FROM AN EXTERNAL SUBROUTINE	C	E	N	0	0	2
105	POP	POP	REMOVE BLOCKS FROM THE FUNCTION STACK	C	E	0	1-2	0	2
106	?	?	GET STATEMENT NUMBER OF LAST ERROR	F	E	N	0	1	12
107	HANG	HANG	HANG A DATA FILE UNTIL FREE	C	E	#	2	0	2
108	SEQ	SEQuential	GET NEXT FULL LOGICAL RECORD	F	E	#	2	1	3
109	REC	RECord	INCREMENT LOGICAL RECORD	F	E	#	2	1	3
110	TIME	TIME	GET THE TIME	F	E	#	1	1	12
111	SEEK	SEEK	SEARCH STRING FOR MATCHING SUBSTRING	F	E	$	2	1	3
112	OLAY	OverLAY	SET UP AN OVERLAY BUFFER	C	E	#	1	0	2
113	DUMP	DUMP	DUMP AN OBJECT PROGRAM TO A DEVICE	C	E	0	1-2	0	2
114	ESEQ	ESEQuential	GET NEXT EMPTY LOGICAL RECORD	F	E	#	2	1	3
115	MCALL	Machine CALL	CALL A MACHINE CODE SUBROUTINE	C	E	E	1-3	0	2
116	TAG	TAG	DECLARE OR GET DISC TAG	C	E	#	2	0-1	3
117	SWAP	SWAP	SWAP OUT A DISKETTE	C	E	#	1	0	2
118	TRACE	TRACE	TRACE PROGRAM EXECUTION	C	E	#	1	0	2
119	BIN	Byte IN	INPUT A BYTE FROM A DEVICE	C	E	A	1-	0	2
120	BOUT	Byte OUT	OUTPUT A BYTE TO A DEVICE	C	E	A	1-	0	2

E. ASCII TABLE

ASCII	DEC	OCT	HEX	CTL	ASCII	DEC	OCT	HEX	ASCII	DEC	OCT	HEX
NUL	ØØØ	ØØØ	ØØ		+	Ø43	Ø53	2B	V	Ø86	126	56
SOH	ØØ1	ØØ1	Ø1	A	,	Ø44	Ø54	2C	W	Ø87	127	57
STX	ØØ2	ØØ2	Ø2	B	-	Ø45	Ø55	2D	X	Ø88	13Ø	58
ETX	ØØ3	ØØ3	Ø3	C	.	Ø46	Ø56	2E	Y	Ø89	131	59
EOT	ØØ4	ØØ4	Ø4	D	/	Ø47	Ø57	2F	Z	Ø9Ø	132	5A
ENQ	ØØ5	ØØ5	Ø5	E	Ø	Ø48	Ø6Ø	3Ø	[	Ø91	133	5B
ACK	ØØ6	ØØ6	Ø6	F	1	Ø49	Ø61	31	\	Ø92	134	5C
BEL	ØØ7	ØØ7	Ø7	G	2	Ø5Ø	Ø62	32	]	Ø93	135	5D
BS	ØØ8	Ø1Ø	Ø8	H	3	Ø51	Ø63	33	↑	Ø94	136	5E
HT	ØØ9	Ø11	Ø9	I	4	Ø52	Ø64	34	_	Ø95	137	5F
LF	Ø1Ø	Ø12	ØA	J	5	Ø53	Ø65	35	`	Ø96	14Ø	6Ø
VT	Ø11	Ø13	ØB	K	6	Ø54	Ø66	36	a	Ø97	141	61
FF	Ø12	Ø14	ØC	L	7	Ø55	Ø67	37	b	Ø98	142	62
CR	Ø13	Ø15	ØD	M	8	Ø56	Ø7Ø	38	c	Ø99	143	63
SØ	Ø14	Ø16	ØE	N	9	Ø57	Ø71	39	d	1ØØ	144	64
S1	Ø15	Ø17	ØF	O	:	Ø58	Ø72	3A	e	1Ø1	145	65
DLE	Ø16	Ø2Ø	1Ø	P	;	Ø59	Ø73	3B	f	1Ø2	146	66
DC1	Ø17	Ø21	11	Q	<	Ø6Ø	Ø74	3C	g	1Ø3	147	67
DC2	Ø18	Ø22	12	R	=	Ø61	Ø75	3D	h	1Ø4	15Ø	68
DC3	Ø19	Ø23	13	S	>	Ø62	Ø76	3E	i	1Ø5	151	69
DC4	Ø2Ø	Ø24	14	T	?	Ø63	Ø77	3F	j	1Ø6	152	6A
NAK	Ø21	Ø25	15	U	@	Ø64	1ØØ	4Ø	k	1Ø7	153	6B
SYN	Ø22	Ø26	16	V	A	Ø65	1Ø1	41	l	1Ø8	154	6C
ETB	Ø23	Ø27	17	W	B	Ø66	1Ø2	42	m	1Ø9	155	6D
CAN	Ø24	Ø3Ø	18	X	C	Ø67	1Ø3	43	n	11Ø	156	6E
EM	Ø25	Ø31	19	Y	D	Ø68	1Ø4	44	o	111	157	6F
SUB	Ø26	Ø32	1A	Z	E	Ø69	1Ø5	45	p	112	16Ø	7Ø
ESC	Ø27	Ø33	1B		F	Ø7Ø	1Ø6	46	q	113	161	71
FS	Ø28	Ø34	1C		G	Ø71	1Ø7	47	r	114	162	72
GS	Ø29	Ø35	1D		H	Ø72	11Ø	48	s	115	163	73
RS	Ø3Ø	Ø36	1E		I	Ø73	111	49	t	116	164	74
US	Ø31	Ø37	1F		J	Ø74	112	4A	u	117	165	75
SPA	Ø32	Ø4Ø	2Ø		K	Ø75	113	4B	v	118	166	76
!	Ø33	Ø41	21		L	Ø76	114	4C	w	119	167	77
"	Ø34	Ø42	22		M	Ø77	115	4D	x	12Ø	17Ø	78
#	Ø35	Ø43	23		N	Ø78	116	4E	y	121	171	79
$	Ø36	Ø44	24		O	Ø79	117	4F	z	122	172	7A
%	Ø37	Ø45	25		P	Ø8Ø	12Ø	5Ø	{	123	173	7B
&	Ø38	Ø46	26		Q	Ø81	121	51	¦	124	174	7C
'	Ø39	Ø47	27		R	Ø82	122	52	}	125	175	7D
(	Ø4Ø	Ø5Ø	28		S	Ø83	123	53	~	126	176	7E
)	Ø41	Ø51	29		T	Ø84	124	54	DEL	127	177	7F
*	Ø42	Ø52	2A		U	Ø85	125	55				

F. TECHNICAL DATA

Lay-Out of OPUS



Notable Memory Locations in OPUS, REV. 2.0

Address Octal	Hex	# Bytes	Description
ØØ2 173	Ø27B	-	Starting address to restart OPUS without losing programs in memory
ØØ2 2Ø2	Ø282	2	Number of bytes in memory (top memory)
ØØ2 2Ø4	Ø284	1	Maximum number of terminals
ØØ2 2Ø5	Ø285	1	Maximum number of discs
ØØ2 2Ø6	Ø286	2	Start of disc parameter table
ØØ2 21Ø	Ø288	2	Start of disc specification buffer
ØØ2 214	Ø28C	1	Number of interrupt routines
ØØ2 215	Ø28D	2	Start of machine stack
ØØ2 217	Ø28F	2	Start of job parameter table
ØØ2 221	Ø291	8Ø	Table of I/O routine addresses
ØØ2 341	Ø2E1	2Ø	Table of initialization routine addresses

The following locations are relative to the start of the job parameter table:

ØØØ	ØØ	2	Size of Operand Table
ØØ2	Ø2	2	Size of Constant Table
ØØ4	Ø4	2	Size of Variable Name Table
ØØ6	Ø6	2	Size of input buffer
Ø1Ø	Ø8	2	Maximum assigned files
Ø12	ØA	2	Number of bytes precision
Ø33	1B	2	Start ASCII program
Ø35	1D	2	Start source program
Ø37	1F	2	Start object program
Ø41	21	2	Start operand stack
Ø43	23	2	Start operand stack block
Ø45	25	2	End operand stack
Ø47	27	2	Start Constant Table
Ø51	29	2	End Constant Table
Ø53	2B	2	Start Variable Name Table
Ø55	2D	2	Start Variable Name Table block
Ø57	2F	2	End Variable Name Table
Ø61	31	2	Start input buffer
Ø63	33	2	Start command buffer
Ø65	35	2	Start file buffers
Ø67	37	2	Start number buffers
Ø77	3F	2	Start Variable Value Table
1Ø1	41	2	End Variable Value Table
1Ø3	43	2	Start of function stack
1Ø5	45	2	End of function stack (top memory)

XVIII. SAMPLE PROGRAMS

A. PROGRAMS WHICH RUN UNDER OPUS/ONE

Bubble Sort

```
10  REM "EXAMPLE OF BUBBLE SORT IN OPUS/ONE"
20  DIM M( 25) ;
30  REM "GENERATE 25 RANDOM NUMBERS"
40  LOOP I, 1TO 25; M( I) = RND ; NEXT ;
50  X= "ORIGINAL ORDER"; GOSUB "MATRIX:";
60  REM "SORT NUMBERS"
70  LOOP I, 1TO 24; LOOP J, I+ 1TO 25;
80  IF M( J) < M( I) [ X= M( I) ; M( I) = M( J) ; M( J) = X] ;
90  NEXT ; NEXT ;
100  X= "SORTED ORDER"; GOSUB "MATRIX:";
110  END ;
120  REM "SUBROUTINE TO PRINT MATRIX"
130  PRINT "MATRIX:", X; LIN 1;
140  LOOP I, 1TO 5;
150  PF "5(R14)/", ( LOOP J, 1TO 5[ M( 5* ( I- 1) + J) ] NEXT 1) ;
160  NEXT ;
170  LIN 2;
180  RETURN ;
```

```
MATRIX:   ORIGINAL ORDER

     .65325928      .86286926      .92886353      .53703308      .12219238
     .15333557      .15316772 1.4511108E-02      .92608643      .97016907
     .20462036      .79179382      .48681641      .54774475      .88009644
     .97825623      .97625732      .87043762      .72647095      .43327332
     .31628418      .67262268 4.0618896E-02      .76622009      .17877197

MATRIX:   SORTED ORDER

 1.4511108E-02 4.0618896E-02      .12219238      .15316772      .15333557
     .17877197      .20462036      .31628418      .43327332      .48681641
     .53703308      .54774475      .65325928      .67262268      .72647095
     .76622009      .79179382      .86286926      .87043762      .88009644
     .92608643      .92886353      .97016907      .97625732      .97825623
```

```
10  REM "THIS CALCULATOR PROGRAM RUNS UNDER OPUS/ONE"
20
30  PRINT "******  CALCULATOR PROGRAM  ******"
40
50  PRINT ; INPUT "DO YOU WANT INSTRUCTIONS? ", X$
60  IF X$ ( 1, 1) IS "Y"
70
80  [ PRINT "THIS PROGRAM WILL KEEP A RUNNING TOTAL OF ARITHMETIC"
90  PRINT "OPERATIONS. AFTER THE PROMPT CHARACTER ':', ENTER ONE OF"
100  PRINT "THE FOLLOWING:"
110  PRINT
120
130  PRINT "<NUMBER>                          SETS TOTAL TO THIS VALUE"
140  PRINT "<BINARY><NUMBER>                   PERFORMS THE BINARY OPERATION USING"
150  PRINT "                                   THE TOTAL AND THE NUMBER"
160  PRINT "<BINARY><UNARY><NUMBER>            PERFORMS UNARY OPERATION ON NUMBER"
170  PRINT "                                   AND THEN BINARY OPERATION WITH TOTAL
180  PRINT
190  PRINT "BINARY OPERATORS:"
200  PRINT "    +    ADDITION"
210  PRINT "    -    SUBTRACTION"
220  PRINT "    *    MULTIPLICATION"
230  PRINT "    /    DIVISION"
240  PRINT "    ^    POWER"
250  PRINT
260  PRINT "UNARY OPERATORS:"
270  PRINT "    NEG    NEGATION"
280  PRINT "    SQR    SQUARE ROOT"
290  PRINT "    SIN    SINE"
300  PRINT "    COS    COSINE"
310  PRINT "    TAN    TANGENT"
320  PRINT "    ATN    ARCTANGENT"
330  PRINT "    EXP    EXPONENTIAL E"
340  PRINT "    LOG    LOGARITHM"] ;
350
360
370  PRINT ;
380  INPUT "ENTER NUMBER DIGITS PRECISION: ", X; X= ( XMAX 2) ; SET 6, X;
390  LIN 2;
400
410  DIM BO( 6) ; SCAN "BINARY": X, BO;
420  DIM UO( 9) ; SCAN "UNARY": X, UO;
430
440  WHILE ( INPUT ": ", X$ NOT NONE ) ;
450
460  LOOP B, 1TO 5* ( X$ ( 1, 1) # BO( B) ) ; NEXT ;
470
480  N= NUM X; IF LEN X> 1AND B< 6[ X= X$ ( 2) ;
490  WHILE X$ ( 1, 1) IS " "AND LEN X> 1;
500  X= X$ ( 2) ;
510  CONT ;
520  N= NUM X;
530
540  IF LEN X> 2[ LOOP U, 1TO 8* ( X$ ( 1, 3) # UO( U) ) ; NEXT ;
```

```
550  IF LEN X> 3AND U< 9[ X= X$ ( 4) ;
560  WHILE X$ ( 1, 1) IS " "AND LEN X> 1;
570  X= X$ ( 2) ;
580  CONT ;
590
600  K= NUM X;
610  N= ( ON U[ SQR K] \ [ - K] \ [ SIN K] \ [ COS K] \ [ TAN K] \ [ ATN K] \
620  [ EXP X] \ [ LOG X] \ [ K] ) ] ] ] ;
630
640  T= ( ON B, [ T+ N] \
650  [ T- N] \
660  [ T* N] \
670  [ T/ N] \
680  [ T^ N] \
690  [ N] ) ;
700
710  SPA 20; PRINT T;
720  CONT ;
730
740  END ;
750
760  REM "BINARY OPERATORS"
770
780  "BINARY", "+", "-", "*", "/", "^", "";
790
800  REM "UNARY OPERATORS"
810
820  "UNARY", "SQR", "NEG", "SIN", "COS", "TAN", "ATN", "EXP", "LOG", "";
```

Loan Payment Calculator

```
LIST
10  REM "**** CALCULATE LOAN PAYMENTS ****"
20  PRINT "************ PROGRAM TO CALCULATE LOAN PAYMENTS ****************";
30  PRINT ;
40  INPUT "TOTAL LOAN AMOUNT? ", A;
50  INPUT "INTEREST RATE? ", I;
60  INPUT "TERM (IN MONTHS)? ", N;
70  INPUT "OUTPUT DEVICE? ", DV;
80  INPUT "IF PRINTER, POSITION PAPER & RETURN ", QQ;
90  I= I/ 100/ 12;
100  P= ( TRU ( ( ( I* A* ( ( ( I+ 1) ^ N) /
110  ( ( ( I+ 1) ^ N) - 1) ) ) * 100) + .5) ) / 100;
120  Y= A;
130  PF DV: "//L5,B4,L7,B4,L8,B4,L9,B4,L7//",
140  "MONTH", "PAYMENT", "INTEREST", "PRINCIPLE", "BALANCE";
150  LOOP X, 1TO N;
160  Q= ( TRU ( ( ( A* I) * 100) + .5) ) / 100;
170  H= P- Q; A= A- H;
180  PF DV: "R5,B4,R4.2,B4,R5.2,B4,R5.2,B4,R6.2/",
190  X, P, Q, H, A;
200  NEXT ;
210  LIN DV: 2;
220  PRINT "TOTAL PRINCIPLE + INTEREST: "& ( P* N) ;
230  PRINT "TOTAL INTEREST: "& ( ( P* N) - Y) ;

FINE
COM

FINE
RUN
************ PROGRAM TO CALCULATE LOAN PAYMENTS ****************

TOTAL LOAN AMOUNT? 6400
INTEREST RATE? 18
TERM (IN MONTHS)? 12
OUTPUT DEVICE? 4
IF PRINTER, POSITION PAPER & RETURN


MONTH     PAYMENT     INTEREST     PRINCIPLE     BALANCE

    1      586.75        96.00        490.75      5909.25
    2      586.75        88.64        498.11      5411.14
    3      586.75        81.17        505.58      4905.56
    4      586.75        73.58        513.17      4392.39
    5      586.75        65.89        520.86      3871.53
    6      586.75        58.07        528.68      3342.85
    7      586.75        50.14        536.61      2806.24
    8      586.75        42.09        544.66      2261.58
    9      586.75        33.92        552.83      1708.75
   10      586.75        25.63        561.12      1147.63
   11      586.75        17.21        569.54       578.09
   12      586.75         8.67        578.08          .01


TOTAL PRINCIPLE + INTEREST: 7041
TOTAL INTEREST: 641

  NE
```

B. PROGRAMS WHICH RUN UNDER OPUS/TWO

Quick Sort

```
10  REM "EXAMPLE OF A QUICK SORT ALGORITHM USING OPUS/TWO"
20  REM "25 RANDOM NUMBERS ARE GENERATED AND SORTED"
30  EXT "PRNT"; EXT "QK"; DIM MAT( 25) ;
40  LOOP I, 1TO 25; MAT( I) = RND ; NEXT ;
50  CALL PRNT( "ORIGINAL ORDER:", MAT) ;
60  CALL QK( MAT, 1, 25) ;
70  CALL PRNT( "SORTED ORDER:", MAT) ;
80  END ;
90
100  REM "SUBROUTINE TO PRINT MATRIX"
110  SUB "PRNT", PRNT, X, M;
120  LIN 3; PRINT X; LIN 1;
130  LOOP I, 1TO 5;
140  PF "5(R14)/", ( LOOP J, 1TO 5[ M( 5* ( I- 1) + J) ] NEXT 1) ;
150  NEXT ;
160  RET ;
170
180  REM "RECURSIVE SUBROUTINE TO SORT A MATRIX"
190  SUB "QK", QK, T, S, E;
200  I= S; J= E;
210  WHILE I# J;
220  IF T( I) > T( J) [ X= T( I) ; T( I) = T( J) ; T( J) = X; F= NOT F] ;
230  IF F[ I= I+ 1] ELSE [ J= J- 1] ;
240  CONT ;
250  IF J+ 1< E[ CALL QK( T, J+ 1, E) ] ;
260  IF I- 1> S[ CALL QK( T, S, I- 1) ] ;
270  RET ;
```

```
ORIGINAL ORDER:

     .66702271      .23271179       .1796875      .92225647      .89718628
 5.1116943E-03      .16131592 1.8386841E-02      .67324829      .92106628
     .36853027       .4690094      .29208374      .58995056      .47320557
     .75849915      .80056763      .37113953       .8972168      .12123108
     .99557495      .37400818      .81243896       .5415802      .42398071


SORTED ORDER:

 5.1116943E-03 1.8386841E-02      .12123108      .16131592       .1796875
     .23271179      .29208374      .36853027      .37113953      .37400818
     .42398071       .4690094      .47320557       .5415802      .58995056
     .66702271      .67324829      .75849915      .80056763      .81243896
     .89718628       .8972168      .92106628      .92225647      .99557495
```

The Maze

```
REM "******  THE MAZE  *****"
REM "THIS PROGRAM RUNS UNDER OPUS/TWO."
REM "IT WAS ENTERED IN THE TEXT EDITOR AND THE LISTING REFLECTS THIS"
REM "FORMAT. IT MAY BE MODIFIED TO RUN UNDER OPUS/ONE BY CHANGING THE"
REM "EXTERNAL SUBROUTINE REFERENCE 'XY' TO A 'GOSUB' SUBROUTINE"

EXT "XY"; PF "S",ASC 26; ESC=(ASC 27)&"=";

PRINT "EXPERIMENT #QZ2002:   TOP SECRET, CONFIDENTIAL!!!^G^G^G
PRINT;

IF (INPUT "IS A DESCRIPTION OF THE EXPERIMENT NECESSARY? ",X;
        X$(1,1) IS "Y")

PRINT;
PRINT "THIS EXPERIMENT HAS BEEN SET UP TO DETERMINE WHETHER A RAT
PRINT "INJECTED WITH INTELLIGENCE POTION #2002 (THIS IS YOU) HAS
PRINT "DEVELOPED TO THE POINT OF BEING ABLE TO OVERCOME THE SUPERIOR
PRINT "MENTALITY OF THE WORLD'S MOST RESPECTED SCIENTIST (THE COMPUTER).
PRINT "TO DISCOVER THE OUTCOME OF THIS QUESTION OF GREAT MAGNITUDE,
PRINT "A MAZE WILL BE BUILT BY THE SCIENTIST. THE SCIENTIST WILL FIRST
PRINT "DISCOVER HIS OWN PATH THROUGH. THE RAT THEN HAS A CHANCE TO DO
PRINT "IT IN LESS TIME (OR FEWER TRIES).
PRINT
PRINT "A SCOREBOARD OF THE NUMBER OF THE RAT'S MOVES (LEFT SIDE)
PRINT "AND THE SCIENTIST'S MOVES (RIGHT SIDE) WILL BE KEPT IN THE
PRINT "RIGHT-HAND CORNER. THE RAT, BECAUSE IT TIRES MORE EASILY, WILL
PRINT "HAVE THE OPTION TO TRY MORE MAZES. A FINAL TABULATION WILL BE
PRINT "GIVEN AT THE END.
PRINT
PRINT "THE SCIENTIST WILL ASK FOR THE RAT'S 'MOVE'. THE RAT SHOULD
PRINT "ENTER ONE OF THE FOLLOWING TO DETERMINE IN WHICH DIRECTION THE
PRINT "PATH SHOULD GO:
PRINT "      U      UPWARDS
PRINT "      D      DOWNWARDS
PRINT "      R      RIGHT
PRINT "      L      LEFT
PRINT
PRINT "THE SCIENTIST WILL INFORM THE RAT WHEN THE CENTER IS REACHED.
PRINT "NOTE THAT IF THE RAT RUNS INTO A BARRICADE, THIS WILL
PRINT "NONETHELESS COUNT AS A MOVE.
PRINT
PRINT "THE FOLLOWING SYMBOLS DEFINE THE MAZE:
PRINT "      @      BARRICADE
PRINT "      *      RAT'S PATH
PRINT "      +      SCIENTIST'S PATH
PRINT "      &      BOTH SCIENTIST'S AND RAT'S PATH
LIN 2 3;

WHILE ( PF "SSSB70-",ESC,ASC 55,ASC 32;
        INPUT "IS THE RAT PREPARED FOR THE NEXT MAZE? ",X;
        NOT NONE AND X$(1,1) IS "Y");
```

```
REM "GENERATE MAZE"
PF "SS/",ASC 26,"THE SCIENTIST IS BUILDING THE MAZE...";

DIM M(23,23);
LOOP I,1 TO 11;
        LOOP J,I TO 24-I;
                M(I,J)=(M(J,I)=(M(24-I,J)=(M(J,24-I)="@")));
        NEXT;
NEXT 2;

REM "CREATE RANDOM GATES AND BARRICADES"
PRINT "THE SCIENTIST SETS UP THE BARRICADES AND GATES...";

LOOP I,2 TO 9;
        R=TRU(RND*(13-I))+1;

        LOOP J,1 TO R;
                CALL XY(I,X,Y);
                M(Y,X)=(IF I/2 IS TRU(I/2) ["/"] ELSE ["-"]);
        NEXT;

NEXT;

REM "CREATE PATH THROUGH MAZE"
PRINT "THE SCIENTIST IS NOW FINDING A PATH THROUGH...";

CTTL=0; C=@XY(1,SX,SY); X=SX; Y=SY; M(Y,X)="+";

LOOP I,2 TO 10;
         CALL XY(I,A,B); D=2*TRU(RND*2)-1;

WHILE (X#A OR Y#B);

        IF C [X=X+D] ELSE [Y=Y+D];
        IF M(Y,X) IS "@" OR M(Y,X) IS "-"
                 [IF C [X=X-D; D=(IF Y<12 [1] ELSE [-1])]
                        ELSE [Y=Y-D; D=(IF X<12 [1] ELSE [-1])];
                C=NOT C]
        ELSE     [M(Y,X)="+"; CTTL=CTTL+1];

CONT;

IF C [Y=Y+1-2*(Y>11)] ELSE [X=X+1-2*(X>11)];
M(Y,X)="+"; CTTL=CTTL+1;

NEXT 2;

M(12,12)="+"; CTTL=CTTL+1;

REM "PRINT MAZE"

PF "S",ASC 26;
LOOP I,1 TO 23; LOOP J,1 TO 23;
        IF (A=M(I,J)) IS "@" OR A IS "/"
```

```
                [PF "SSSS",ESC,ASC(I+31),ASC(J*2+30),"@"];

NEXT; NEXT;
PF "SSSS",ESC,ASC 43,ASC 54,"!";

REM "PLAYER ATTEMPT"

PTTL=0; X=SX; Y=SY; TRY="UDLR?"; M(Y,X)="&";
PF "SSSS",ESC,ASC(Y+31),ASC(X*2+30),"*";

WHILE (X#12 OR Y#12);

PF "SSSR4R6",ESC,ASC 32,ASC 92,PTTL,CTTL;

WHILE (PF "SSSSS",ESC,ASC 42,ASC 90,"RAT MOVES: "; A=0; INPUT A;
        LOOP I,1 TO 4*(TRY$(I,I)#A$(1,1)); NEXT;  I>4);
        PTTL=PTTL+1;
CONT;

TY=Y; TX=X;
ON I    [TY=Y-1]\[TY=Y+1]\[TX=X-1]\[TX=X+1];

IF TY>0 AND TY<24 AND TX>0 AND TX<24
        [IF (J=M(TY,TX))#"@" AND J#"/"
                [Y=TY; X=TX; M(Y,X)=(IF J#"+" ["*"] ELSE ["&"]);
                PF "SSSS",ESC,ASC(Y+31),ASC(X*2+30),"*" ]];

PTTL=PTTL+1;
CONT;

PF "SSS",ESC,ASC 55,ASC 32;
PF "SB5S",
(ON (A=(SGN(CTTL-PTTL)+2)) THEN
        "POTION #2002 IS FAILING..."\
        "POTION #2002 NEEDS PERFECTING..."\
        "THE SCIENTIST IS DISBELIEVING..."),
        "HERE IS THE SCIENTIST'S ATTEMPT";

PGAME=PGAME+PTTL; CGAME=CGAME+CTTL; TGAME=TGAME+1;

LOOP I,1 TO 23; LOOP J,1 TO 23;
        IF (A=M(I,J)) IS "+" OR A IS "&"
                [PF "SSSS",ESC,ASC(I+31),ASC(J*2+30),A];
NEXT; NEXT;
PF "SSSS",ESC,ASC 43,ASC 54,"!";

CONT;

PF "SSR4B2S//",ASC 26,"EXPERIMENT SUMMARY FOR",TGAME,"ATTEMPTS";

PF"B2OL9R11//","SCIENTIST","RAT"; X="L2OR9B2R9/";
PF X,"TOTAL MOVES",CGAME,PGAME;
PF X,"AVE/ATTEMPT",CGAME/TGAME,PGAME/TGAME;
PF X,"IQ",200-CGAME/TGAME,200-PGAME/TGAME;
```

```
PRINT;
ON SGN(CGAME-PGAME)+2 PRINT
        "POTION #2002 IS NO GOOD AND MANKIND MAINTAINS SUPERIORITY"\
        "IMPOSSIBLE RESULTS. EXPERIMENT MUST BE RE-DONE"\
        "POTION #2002 WILL CHANGE THE WORLD!!!";

END;


REM "SUBROUTINE TO GENERATE RANDOM COORDINATES AT GIVEN LEVEL"
REM "L IS MATRIX LEVEL 1-12"
REM "X IS GENERATED COLUMN"
REM "Y IS GENERATED ROW"
REM "RETURN C=0 IF VERTICAL CONROL, C=1 IF HORIZONTAL CONTROL"

SUB "XY",XY,L,X,Y;

A=TRU(RND*(21-TRU(L/2)*4))+2+TRU(L/2)*2; B=TRU(RND*4)+1;

ON B    [X=A; Y=L; C=1]\
        [X=L; Y=A; C=0]\
        [X=A; Y=24-L; C=1]\
        [X=24-L; Y=A; C=0];

RET C;
```

XIX. INDEX

ORDER FORM

QTY.	#	ITEM	MEDIUM	PRICE
____	D-1a	OPUS/ONE Language, Disc Version	_____	_____
____	D-1b	OPUS/ONE Language, Cassette/Paper Tape Version	_____	_____
____	D-1d	OPUS/TWO Language, Disc Version	_____	_____
____	D-1x	Change starting location of OPUS/ONE or OPUS/TWO Language (normal starting address: ØØØ) to_____	_____	_____
____	D-1z	FORTE/TWO Monitor	_____	_____
____	D-1m	OPUS User's Manual		_____
		**********		
____	U-1	Upgrade OPUS/ONE to OPUS/TWO	_____	_____
____	U-2	Copying Charge	_____	_____
		**********		
____	D-2	TEMPOS Multi-User/Multi-Tasking Operating System Includes: Assembler, Text Editor, Utilities Programs, and OPUS High-Level Language	_____	_____
____	D-2m	TEMPOS User's Manual		_____
		**********		
____	A-1a	Clinical Accounts Receivable/Billing System	_____	_____
____	A-1b	Clinical Accounts Receivable/Billing Source	_____	_____
____	A-1d	Clinical Accounts Receivable/Billing Demo Package	_____	_____
____	A-1m	Clinical Accounts Receivable/Billing User's Manual		_____
		**********		
____	M-1	PROM Loader for _____ (software)	1702A I.C.	_____
____	M-2	Billing Forms		
		**********		
		SUBTOTAL.		_____
		Colorado Residents ADD 3% Sales Tax		_____
		SHIPPING for prepaid and bank card orders (ADD: $1.00 per manual or $2.00 per language medium ordered)		_____
		TOTAL		_____

☐ My check or money order for the full amount is enclosed
☐ C.O.D. (C.O.D. fee will be added) -- UNITED STATES ONLY
☐ Please bill my ☐ Master Charge ☐ VISA (BankAmericard) Account:

Account No._____ Expiration Date_____
Master Charge Interbank #_____

Signature_____
(We must have your signature to process credit card sales.)

PLEASE SEND THE ABOVE ITEMS TO:

NAME_____

COMPANY_____

ADDRESS_____

CITY_____ STATE_____ ZIP_____

MY HARDWARE CONFIGURATION:

Please allow 30 days from receipt of order for processing.

Administrative Systems, Inc. — 222 Milwaukee, Suite 102, Denver, Colorado 80206 (303) 321-2473