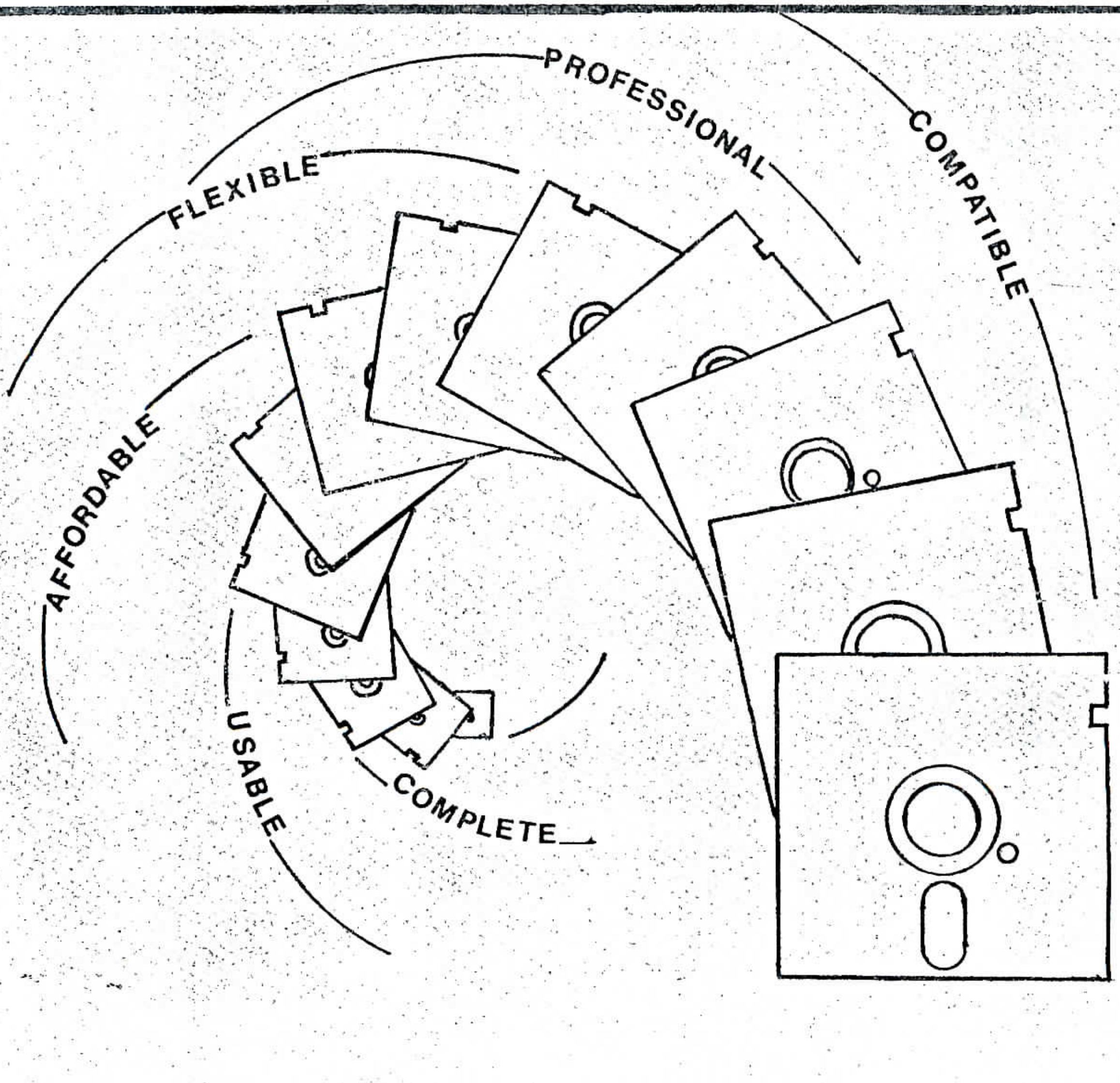
REFERENCE MANUALS



Optimized Systems Software

GETTING STARTED WITH OSS

CONGRATULATIONS !!!

You have purchased what we believe is by far the most advanced software development package available for the Atari 800 and Atari 400 personal computers.

This package will run on any Atari 800 or Atari 400 with at least 32K bytes of RAM. Since no OSS software uses any routines in any cartridge, you may fully utilize all the RAM in even a 48K byte Atari.

CAUTION: If you have ANY cartridge plugged into your Atari, you will not be able to utilize more than 40K bytes of memory. This a hardware feature of the Atari and can not be changed via software. If you need the power of 48K bytes of RAM, REMOVE ALL CARTRIDGES.

There are, however, some circumstances under which you may need a cartridge for your own development work. OSS CP/A is completely compatible with all known Atari cartridges.

HOW TO USE YOUR OSS PACKAGE

1. Check the contents of your package. If you ordered just BASIC A+, there should be a BASIC A+ manual (an addendum to the Atari Basic manual). If you ordered CP/A, there should be a CP/A manual and an EASMD (Editor/ASseMbler/Debug) manual.

2. There should be a license agreement. FILL THIS OUT NOW AND SEND IT TO US ! Aside from its obvious purpose, the agreement is YOUR ticket to ***SUPPORT***. Yes, we do answer phone questions. Yes, we do respond to bugs. BUT ONLY for those persons who send back their license !!!

3. Turn on your disk drive(s) and screen, leave the Atari computer off. If you purchased CP/A, place the CP/A disk in drive 1. If you purchased only BASIC A+, place an Atari DOS master disk in drive 1. Boot up the system by turning on the computer's power. That's all there is to it! Follow the manual directions for running the program you desire.
   Note: special instructions for running BASIC A+ under Atari DOS are in the beginning of the BASIC A+ manual.

4. We strongly urge you to immediately make a backup copy of your OSS diskette. You may do this using DUPDSK (see CP/A manual). [Or use the Atari DUPLICATE DISK menu DOS command.]

5. Sit back and enjoy the power of a REAL computer system.

ABOUT ERRORS

Since this is a NEW product, there are bound to be a few bugs lurking in the cracks. We hope and believe that such bugs as there might be will be mere annoyances. PLEASE report any bugs or suspected bugs to OSS as soon as possible. Unless you are absolutely desparate, PLEASE document the bug IN WRITING and include an example program. We WILL accept diskettes which demonstrate problems, particularly complex ones.

ERRATA

The following are known bugs. You may expect user notes and/or updates regarding fixes for these bugs in the near future. PLEASE send in your registration form, or we cannot contact you!

BASIC A+

1. LVAR

When LVAR is used with a file or device other than "E:" (i.e., the screen), the device is not closed properly and further printing will take place to the device instead of to the screen. Several statements will halt the erroneous process, but the simplest way is:

```
LVAR "P:" : ?
```

The list of variables will be sent to the printer and the '?' (abbreviation for PRINT) statement will print one extra blank line before restoring normal screen operations.

2. PRINT USING

When an arithmetic expression is to be printed via PRINT USING, if the expression contains arithmetic involving multiplication or division (including, therefor, all transcendental functions), the system will hang and will require SYSTEM RESETting. EXAMPLE:

```
PRINT USING " ###### ", A * 3
```

will hang because of the multiply of A times 3. SOLUTION:

```
TEMP=A * 3 : PRINT USING " ###### ", TEMP
```

Simply don't use * or / in arguments to Print Using.

3. The manual

Figure PMG-1 from the page numbered 74 in your BASIC A+ manual is missing. The replacement Table of Contents is missing. A new page 74 and the Contents pagesare attached to this ERRATA section.

NOTICE

OSS reserves the right to make changes or improvements in the product described in this manual at any time and without notice.

Optimized Systems Software
10379 Lansdale Avenue
Cupertino, CA 95014
Telephone: 408-446-3099

OPTIMIZED SYSTEMS SOFTWARE

CP/A

for the ATARI 800 (R)

MARCH 1981

Version 1.0

TABLE OF CONTENTS

1.0 Introduction

CP/A is an abbreviation for Command Processor/Advanced. Its purpose is to give its user an advanced type of command control over the software systems in the Atari personal computer. CP/A replaces the Atari menu driven DOS command processor with a less restrictive command line processor. The CP/A user types command words and parameters rather then invoking menu functions and responding to questions. The CP/A command set is easy to learn since most of the commands are the same as the functions desired, such as RENAME or DIRECTORY. The CP/A command processor allows for user written commands as well as the "batch" execution of commands from a file. CP/A replaces ONLY the MENU command processor of ATARI DOS. The Atari File Manager and Atari OS are used by CP/A without modification. This means that disk volumes and associated disk files are fully interchangable between Atari DOS and CP/A. The only known incompatibility is that OSS BASIC A+ SAVE files are not compatable with ATARI BASIC SAVE files. ATARI BASIC ATASCII source (LIST, ENTER) files will run without modification under OSS BASIC A+. The DOS.SYS file on the CP/A disk is the Atari FMS (written by OSS) and CP/A. CP/A disks do not have or need to have DUP.SYS or MEM.SAV.

2.0 Running CP/A

The CP/A Command Processor is invoked in the same manner as the Atari menu command processor. When the CP/A disk is booted, CP/A is immediatly entered. If the computer has a cartridge that works with the disk, such as BASIC, then the cartridge can be entered via the CP/A CARtridge command. Re-entry of CP/A from the cartridge is done in the same way as it is to Atari DOS. The BASIC command for this is DOS. Some cartridges do not allow DOS type exits and thus CP/A cannot be used with these cartridges.

When CP/A is entered it will clear the screen and display:

```
        OSS CP/A   ATARI version 1.0
        Copyright (c) 1981  OSS

        D1:<cursor>
```

The D1: is the command prompt. It serves two purposes. First it tells the user it is ready to accept a command. Secondly, it is a reminder of the default disk drive. The default drive, in this case, being the familar file spec for drive 1.

3.0 Default Drive and File Specs:

Most CP/A commands and parameters deal with files of one sort or another. The Atari Operating System requires files be specified with a filespec of the form:

```
<device>: <optional-file-name>
```

The device for disk files is of the form Dn: where n=1,2,3,4. For example, D1: is the device name of the disk drive with the switch at the rear of the drive set for drive one. Other types of devices are: Printer=P:, Cassette=C:, Screen=S:, etc. The optional-file-name is used for named file accessing devices such as the disk units. To work with the disk file TEST.ORG on disk drive number 1, the operating system requires that the file spec D1:TEST.ORG be used. Having to always specify the D1: can be tedious, especially if most of the user's file work is on a single drive.

The CP/A system is designed to prefix all filenames appearing in a CP/A command line with the default drive - if a device has not been explictly specified. In the case of D1:TEST.ORG, the user could enter only TEST.ORG for a file name and allow CP/A to prefix it with the default drive. Thus TEST.ORG becomes D1:TEST.ORG in the CP/A system. If TEST.ORG happened to be on drive two and the default drive was drive one, the user could enter D2:TEST.ORG. CP/A would see that the user has explicitily specified a <device> and would thus not append the default drive device to that file name.

If the user needs to work a great deal with files on drive two, he can change the default drive so as to avoid the now necessary D2: prefix typing. Where the system prompts D1:<cursor>, the user can respond with D2:<return> to change the default drive to the D2: device. The next CP/A prompt line will show D2:<cursor>. Now files accessed on drive one will require the explict D1: prefix typing, while files on drive will not require prefix typing. Only devices of the form Dn: (where n = 0-9) are allowed as default drives. CP/A does not check to insure that the new default drive actually exists. The user's first indication of an invalid default drive will occur when CP/A attempts to access a file on the invalid device (via user command). The error message "INVALID DEVICE" will indicate the situation. The user should then set the default device to a valid disk unit. The default device change command is one of the many CP/A commands.

4.0 CP/A Commands

CP/A has three general classes or groups of commands. The classes are intrinsic commands, extrinsic commands, and execute commands. Intrinsic commands are executed by means of resident code in the CP/A monitor. Extrinsic commands are executed by means of loading and running programs. The execute subset of commands provide for the batch execution of CP/A commands from a file.

4.1 Intrinsic Commands:

The intrinsic commands are executed via code in the CP/A monitor. These commands do not require the loading of programs to perform their functions. The following is a summary of the CP/A intrinsic commands:

```
DIRECTORY - List Directory
PROTECT   - Protect a file (from change or erase)
UNPROTECT - Unprotect a file
ERASE     - Erase (delete) a file
RENAME    - Renames a file
LOAD      - Load a binary file
SAVE      - Save a binary file
RUN       - Execute a program at some address
CARTRIDGE - Run Atari cartridge in the A cartridge slot
```

The default drive change command Dn: is also an intrinsic command. All intrinsic commands may be abreviated with the first three characters. As a matter of fact, CP/A only looks at the first three characters while testing for an intrinsic command. Each of the commands will be covered in detail later in this manual; however, to give you a feel of the intrinsic commands let's look at the DIRECTORY command. While looking at these examples, assume the D1: is the default device and has been placed on the screen by CP/A.

```
D1:DIRECTORY    list entire directory of disk on drive one
D1:DIRECT        "     "       "      "   "    "    "    "
D1:DIRTY         "     "       "      "   "    "    "    "
D1:DIR           "     "       "      "   "    "    "    "
D1:DIR *.*       "     "       "      "   "    "    "    "
D1:DIR D1:       "     "       "      "   "    "    "    "
D1:DIR D1:*.*    "     "       "      "   "    "    "    "
D1 DIR D2:      list entire directory of disk on drive two
D1:DIR D2:*.*    "     "       "      "   "    "    "    "
D1:DIR *.OBJ    list all files with extension .OBJ on drive one
D1:DIR D2:*.ASM list all files with extension .ASM on drive two
```

4.1.1 PROTECT

The PROtect command is used to protect disk files from being modified or ERAsed. Files that have an asterisk to the left of the file name in the directory listing are protected files.

```
                PROtect         file spec
```

4.1.2 UNPROTECT

The specified files (PROtected or not) are unprotected. The unprotected files can now be modified or ERAsed.

```
                UNProtect       filespec
```

4.1.3 ERASE

The specified files are removed from the disk and the disk sectors occupied by the files become free to be used again by other files.

```
                ERAse           filespec
```

4.1.4 RENAME

Rename a file or files.

```
                REName  old-filespec new-filename
                REName  old-filespec,new-filename
```

The old-filespec specifies the file(s) that are to be renamed to new-filename. Either blanks or a comma may be used to separate the filenames. WARNING! Be careful using wild card renames. You can get more than one file with the same name and never be able to access the second same-named file. (See Appendix B)

4.1.5 SAVE

The SAVE command is used to write (copy) a section of RAM to a disk file. The area of RAM to be written is given as the two hexidecimal parameters start address (sa) and end address (ea).

```
                SAVe    filespec  sa   ea
        Example:
                SAV     TEST.OBJ  4000  4FFF
```

The sa and ea parameters are separated by blanks or a comma. The ea must be greater than or equal to sa.

CP/A will write a six byte header to the file before writing the data. This header consists of the binary file indicator, the sa, and the ea.

Binary File Indicator (2 Bytes)		$FFFF
sa	(2 Bytes) least significant byte first	($0040)
ea	(2 Bytes) least singificant byte first	($FF4F)
data	(ea - sa) + 1 bytes	

The saved file may be later loaded with the LOAD command.

4.1.6 LOAD COMMAND

The Load command is used to load binary files into RAM. The specified file is checked for the Binary File Indicator ($FFFF as the first two file bytes). If the indicator is present the next four bytes are assumed as the sa and ea of the data. CP/A will then copy the next ea-sa + 1 bytes of data from the file to RAM starting at ea. CP/A will also place sa in the CP/A RUNLOC cell. If CP/A does not recieve an end-of-file after loading the data it will assume another code segment is present. Each following code segment is like the first except that the $FFFF header is not present. CP/A will only place the sa from the first segment in RUNLOC.

```
        LOAD    filespec
```

CP/A also supports the Atari load and go scheme. If the load file has the proper INIT and RUN vectors, CP/A will perform the INIT and RUN functions (see Atari DOS 2.0 manual for details).

The OSS assembler (EASMD.COM) creates object files that are loadable as LOAD files.

4.1.7 RUN COMMAND

The Run command causes CP/A to call (JSR) a routine in RAM.

```
        RUN     optional-hex-address
```

If the optional hex adress is specified then CP/A will place the given hex adress in the CP/A RUNLOC and then call the routine via the address in RUNLOC. If the hex address is not specified then CP/A will call the address that is currently in RUNLOC. The address in RUNLOC may have been set by a previous LOAD or RUN command or via the execution of an extrinsic command.

4.1.8 CARTRIDGE

The parameterless CARtridge command causes CP/A to transfer control to the CARTRIDGE in the A cartridge slot. There are two ways CP/A will call a cartridge, either with a warm start or a with a cold start. The cartridge cold start tells the cartridge to reinitialize its memory and start cold. The warm start tells the cartridge to retain its memory as it was upon exit (via DOS command or RESET). The first CP/A cartridge call will always be a cold start. Subsequent CP/A calls will be warm starts unless CP/A has executed a memory changing command. Memory changing commands are LOAD and extrensic commands.

4.1.8.1 RESET

If a cartridge has control and the RESET key is pressed, CP/A will be entered. If it is desired to re-enter the cartridge, simply enter the CAR command.

4.2 Extrinsic Commands:

The extrinsic commands are programs that are run by CP/A. Any program file of the load file format and containing the .COM extension may be used as a CP/A extrinsic command. The CP/A COPY command is one such extrinsic command. If you DIR the CP/A diskette, you will see a file named COPY.COM . The program in the COPY.COM file is what is executed when the COPY command is entered. Assuming that D1: is the default device, the COPY command would look like:

```
        D1:COPY <from-file-name> <to-file-name>

                          or

        D1:COPY TEST.OBJ D2:TEST.OBJ
```

to copy TEST.OBJ from drive one to drive two.

Whenever any command is given to CP/A it first compares the command entered (first three characters only) to its intrinsic command list. If the command is not in the intrinsic list, it is assumed to be extrinsic. CP/A will process the extrinsic command by:

1) Prefix the command with the default device (if a device is not specified).
2) Attach the .COM extension to the command.
3) Open the generated file spec for input.
4) Test file for proper Load file format (see 4.1.6).
5) Load and execute the program.

The COPY command illustrated will execute only if the file COPY.COM exists on drive one and is of the Load file format. If any element of the procedure fails various error messages will result. Step 1 of the procedure implies that a device may be specified. If the default device is drive two and the COPY.COM program is on drive one, our example COPY would look like:

```
        D2:D1:COPY D1:TEST.OBJ TEST.OBJ
```

which again copies TEST.OBJ from device one to device two. Never explictly specify the .COM extension as part of the command. The command COPY.COM will result in a file spec of D1:COPY.COM.COM, which is invalid. If the file is not of the proper format, the error message ADR RANGE ERROR will most likely appear.

The extrinsic command class contains an infinite number of commands. Some extrinsic commands (such as COPY) are supplied by OSS. Most extrinsic commands are user written. If you are intrested in writing your own extrinsic commands, see Appendix B.

4.3 Batch Processing:

The CP/A execute feature allows the user to execute one or many CP/A commands with a single command. Let's suppose that you wrote a set of BASIC programs that must be run in sequence. You could issue the CP/A extrinsic BASIC command (execute BASIC.COM), then from BASIC run each program one at a time. If the running time of the BASIC programs was very long you could sit at the key board for hours just to type RUN program-name every once in awhile. CP/A allows you to create and execute an EXECUTE file which contains one or many CP/A commands. You would then enter one command that would free you from the keyboard for more important (or fun) things.

4.3.1 Executing EXECUTE files:

Any text file with the filename extension .EXC can be used as a CP/A execute file. The execution of the file is invoked much like the extrinsic commands, except the command is preceeded with an AT "@" symbol. To execute the EXECUTE file DEMO.EXC on the D1: default device

```
        D1:@DEMO
```

CP/A will build the file spec D1:DEMO.EXC and read that file line by line executing the CP/A commands just as if they were being entered from the keyboard.

Humans are not quite perfect in the eyes of computers and sometimes make mistakes. CP/A commands specified in error will generate error messages. If CP/A discovers an error while executing an EXECUTE file, it will print the error message as usual and STOP executing the EXECUTE file.

Execution of an exexute file will also stop after the CARTRIDGE command is executed.

4.3.2 Execute File Format

An execute file is simply a text file. Each line of the text file will become a CP/A command when executed. The three basic rules of text file LINES are that:

1) they must contain valid CP/A commands,
2) they must be less than 60 characters in length
3) they must end in a carriage return (ATASCII $9B).

CP/A allows the commands in an execute file to be preceeded by numbers and blanks. This feature allows the command lines to be numbered for readability and thus document their purposes.
The execute file line:

```
        LOAD    OBJ.TEST <return>
```

and the line:

```
        100 LOAD   OBJ.TEST <return>
```

are the same to CP/A . CP/A scans the line for the first non-numericl, non-blank character before starting to scan the command word. The EDITOR of the OSS EASMD program can be used to create and modify execute files.

4.3.3 Execute Intrinsic Commands

CP/A has four special intrinsic commands designed for use with execute files. These commands are:

```
        REMARK          Remark or comment (does nothing)
        SCREEN          Allows execute commands to echo to the
                        screen. This is the default mode.
        NOSCREEN        Turn off Echo of execute file command lines
                        to the screen.
        END             Stop executing the execute file and return
                        CP/A to keyboard entry mode.
```

4.3.3.1 REMARK

The REMARK command provides a means of commenting and documenting an execute file. CP/A will ignore all characters on the REMARK command line and proceed to the next command file line. The command file:

```
        100     REM     BACKUP DAILY TRANSACTION FILE
        110     BASIC   TFBACKUP.BAS
        120     REM     PRINT TRANSACTION REPORTS
        130     BASIC   TREPORTS.BAS
        140     END
```

uses OSS BASIC A+ to work with some transaction BASIC programs. The REMARK statements explain the process. LINE 110 will load and execute the OSS BASIC A+ (BASIC.COM on default drive) which will in turn run the TFBACKUP.BAS BASIC A+ program (SAVED on default drive).

4.3.3.2 SCREEN/NOSCREEN

CP/A normally echos the command lines to the screen so that it appears as if they were typed in as keyboard commands. The NOSCREEN command can be used to prevent the echo process. After NOSCREEN has been executed, no further EXECUTE file command will appear on the screen until:

1) the SCREEN command is executed or
2) the EXECUTE file stops for some reason.

4.3.3.3 END

The END command provides a documentable END to the execution of of an execute file. It may also be used to stop the file's execution before the actual end-of-file.

4.3.4 PROGRAM CONTROLLED EXECUTE FILE STOP

It is sometimes desirable for a program in a chain of executing programs to stop the execute process. The usual reason for this is that the program has detected an error severe enough to invalidate the processes performed by the following program(s). The continued execution of the execute files is provided for by a single byte flag within CP/A. If a program sets this byte to zero, then upon returning to CP/A via DOS or CP (BASIC statements) the execute file execution will immediately stop. The execute flag is located 11 bytes from the start of CP/A. The address of CP/A is pointed to by memory location 10 ($0A). The following BASIC A+ program segment will turn off the execute file and return to CP/A.

```
1000    CPADR = DPEEK(10)
1010    EXCFLG = CPADR + 11
1020    POKE  EXCFLG,0
1030    DOS
```

4.3.5 STARTUP.EXC

The execute filename STARTUP.EXC has special meaning in the CP/A system. When the system is first booted (power up), CP/A will search the directory of the booted disk volume for a file named STARTUP.EXC. If STARTUP.EXC is on the booted volume, CP/A will execute that file before requesting keyboard commands.

5.0 SYSTEM INTERFACE GUIDE

The writer of assembly language code will most likely need to interface with the Atari Operating System (OS). If the assembly code is to become an extrensic command, there may be a need to interface to CP/A . The Atari OS manual provides a proliferation of information about the Atari Operating System which will not be covered here.

5.1 SYSEQU.ASM

Every CP/A master disk contains an assembler source file, SYSEQU.ASM, that has various commonly used Atari OS and CP/A system equates. This file may be included in an assembly languae program via the OSS EASMD include function (.INCLUDE #D1:SYSEQU.ASM)

5.2 CP/A MEMORY LOCATION

CP/A is designed to be placed just after the Atari File Manager. Since the acatual location of CP/A may vary with different versions of a file manager, a fixed location has been assigned to point to CP/A. The location (CPALOC=$0A) is the same one Atari uses to point to DUP. All Atari programs that use a DOS exit vector through $0A.

5.3 EXECUTE PARAMETERS

The CP/A execute flag is located CPEXFL ($0B) from the start of CP/A. The CPALOC may be used as an indirect pointer to access the execute flag.

```
        LDY     #CPEXFL         ;GET DISPL TO FLAG
        LDA     (CPALOC),Y      ;LOAD FLAG
```

The Execute Flag has four bits that control the execute process:

EXCYES	$80	If one, an execute is in progress
EXCSCR	$40	If one, do not echo execute input to screen
EXCNEW	$10	If one, a new execute is starting. Tells CP/A to start with first line of the file
EXCSUP	$20	If one, a cold start execute is starting. Used to avoid file-not-found error if STARTUP.EXC is not on boot disk.

CP/A performs the execute function by opening the file, POINTing to the next line, reading the next line, NOTE the new next line and closing the file. To perform these functions, CP/A must save the execute file name and the three byte NOTE values. The filename is saved at CPEXFN ($0C) into CP/A. The three NOTE volues are saved at CPEXNP($1C) into CP/A. (CPEXNP+0=ICAUX5; CPEXNP+1=ICAUX4; CPEXNP+2=ICAUX3). By changing the various execute control parameters, a programmer can cause recursion and/or changing of execute files.

5.4 DEFAULT DRIVE

The CP/A default drive file spec is located at CPDFDV ($07) into CP/A. The Default Drive here is ATASCII Dn: where 'n' is the ATASCII default drive number.

5.5 EXTRINSIC PARAMETERS

The extrinsic commands may be called with parameters typed on the command line. The OSS command

```
        D1:COPY FROMFILE  D2:TOFILE
```

is an example of this. The entire parameter line is saved in the CP/A input buffer located at CPCMDB ($40) bytes into CP/A and is available to the user. Since most command parameters are file names, CP/A provides a means of extracting these parameters as filenames. The routine that performs this service begins at CPGNFN ($03) bytes into CP/A . The routine will get the next parameter and move it to the filename buffer at CPFNAM ($21) bytes in CP/A. If the parameter does not contain a device prefix, then CP/A will prefix the parameter with the default drive prefix. The first time COPY calls CPGNFN the file spec "D1:FROMFILE" is placed at CPFNAM. The second time COPY calls CPGNFN the file spec "D2:TOFILE" is placed in CPFNAM. If CPGNFN were to be called again then the default filespec would be set into CPFNAM at each call. To detect the end of parameter condition, the user may check the CPBUFP ($0A into CP/A) cell. If CPBUFP does not change after a CPGNFN call then there are no more parameters. The filename buffer is always padded to 16 bytes with ATASCII EOL ($9B) characters. The following example sets up a vector for calling the get-filename routine.

```
        CLC
        LDA     CPALOC          ;ADD CPGNFN
        ADC     #CPGNFN         ;TO CPALOC VALUE
        STA     GETFN+1         ;AND PLACE IN
        LDA     CPALOC+1        ;ADDRESS FIELD
        ADC     #0              ;OF JUMP
        STA     GETFN+2         ;INSTRUCTION
        ...
        ...
GETFN   JMP     0
```

The following routine then gets the next file name to CPFNAM.

```
        LDY     #CPBUFP         ;SAVE CPBUFP
        LDA     (CPALOC),Y      ;VALUE
        PHA
        JSR     GETFN           ;GET NEXT FILE PARM
        LDY     #CPBUFP
        PLA                     ;TEST FOR NO NEXT
        CMP     (CPALOC),Y      ;PARM
        BEQ     NONEXT          ;BR IF NO NEXTPARM

        LDY     #CPFNAM         ;ELSE GET FILE
        LDA     (CPALOC),Y      ;NAME FROM BUFFER
        ...
        ...
```

5.6 RUNLOC

The CP/A RUNLOC ($3D into CP/A) is used as the CP/A vector to routines with the RUN, LOAD and extrinsic commands. An application that allows exits to CP/A can change RUNLOC to provide a warmstart re-entry to the application (if the user enters RUN with no para-lmeters). If the application is not reusable and wishes to forbid re-entry, the high order byte of RUNLOC ($3E into CP/A) should be set to zero.

```
        LDY     #RUNLOC+1          ;FORBID RE-ENTRY
        LDA     #0                 ;TO ME
        STA     (CPALOC),Y
```

5.7 EXITS

CP/A calls all programs (except cartridges) via the 6502 JSR instruction. A called CP/A program may return back to CP/A via the RTS instruction or via a JMP (CPALOC). If the JMP (CPALOC) is used, CP/A will close IOCB zero and re-open it to the E: device (which clears the screen). Either the JMP (CPALOC) or the RTS return will cause CP/A close IOCBs one through seven.

APPENDIX A

OSS EXTRINSIC COMMANDS

A-1. COPY

```
COPY                from-file-spec            to-file-spec
```

The copy command will copy one file, the from-file-spec, to the to-file-spec. COPY does NOT allow a change of diskettes while copying: both source and destination must be mounted when the COPY command pauses after loading.

A-2. INIT

```
INIT                (no parameters)
```

The INIT command is used to:

1) FORMAT A DISK (OR)
2) FORMAT A DISK AND WRITE DOS.SYS (OR)
3) WRITE DOS.SYS

INIT is menu driven and will give you the oportunity to change disks before executing. DOS.SYS is the CP/A boot loader and is required to make the CP/A boot from the disk.

A-3. DUPDSK

```
DUPDSK              (no parameters)
```

DUPDSK is used to duplicate an entire disk. It can be used with a single drive. It will format the destination disk for you if you choose to do so. When you are finished with DUPDSK, you must insert a system disk (a disk with DOS.SYS) because DUPDSK will (purposefully) re-boot the system.

APPENDIX B

FMS POKES

There are several 'pokes' that can be done to the Atari FMS that comes with CP/A. These pokes are used to change certain FMS parameters. The changes can be made permanent by using INIT to write (re-write) DOS.SYS after the poke is done.

B.1 NUMBER OF FILE BUFFERS

The FMS allocates space for file buffers. One file buffer is required for each open disk file. The number of file buffers allocated is the number of files that can be open at the same time. The CP/A system is shipped with three (3) file buffers allocated. Three is the recommended minimum. The nu1). can be changed by poking a new value at $709 (decimal 1801). The maximum usable value is 7 (any value greater than 7 wastes space). The changed value does not go into effect until the system is booted. This means that you MUST rewrite (or write) DOS.SYS on the disk and then reboot the disk.

B.2 NUMBER OF DRIVES

The FMS drive byte is used to tell FMS how many drives you have on your system. The FMS is shipped with the drive byte set for two drives (D1: and D2:). Each drive allocated via this value consumes an additional 128 bytes of RAM for a drive buffer. If you have more or less than 2 drives, you will probably want to change this value. This value, like the value for the number of file buffers, does not go into effect until the system is booted. The drive byte is located at location $70A (decimal 1802). The appropriate values are:

```
        1 drive = $01   (decimal 1)
        2 drives= $03   (decimal 3)
        3 drives= $07   (decimal 7)
        4 drives= $0F   (decimal 15)
```

B.3 FAST DISK WRITE

The Atari disk can be commanded to write sectors with verify or without verify. The write WITH verify causes the drive to read each sector immediately after writing it; this process assures that data on the disk is valid but causes write operations to run about half as fast as they could run if the write was done without verify. Depending upon your patience, the importance of your data, and your objective view of the reliability of your drives and disks, you can choose either write-with-verify (slow) or write-without-verify (fast). The FMS location to change is $779 (1913 decimal). The write-with-verify value is $57 (87 decimal). The write-without-verify (default, faster write) is $50 (80 decimal).

B.4 RENAME WOES

If you happen to rename several files (for example, with the use of a wild card rename) in such a way that you end up with two files of the same name, you need to remember this section. The problem: after ending up with two files of the same name, all further accesses to that filename will access only the first file that appears in the directory. Even a wildcard rename will not work: both files are again renamed to the same name.

The solution: You may patch FMS to alter the RENAME code. The patch causes RENAME to change only the first file in the directory that matches the given filespec, not all matching filenames. To make the patch, POKE a zero ($00) to location $C2E (decimal 3118). To restore RENAME to normal functioning, poke $B8 (decimal 184) to the same location.

CAUTION: because this patch affects ALL renames, and will not now allow multiple RENAMEs, etc., it is probably not advisable to make the patch permanent.

NOTICE

OPTIMIZED SYSTEMS SOFTWARE reserves the right to make changes or improvements in the product decribed in this manual at any time and without notice.

Optimized Systems Software
10379 Lansdale Ave.
Cupertino CA. 95014

(408) 446-3099

OPTIMIZED SYSTEMS SOFTWARE

OSS EASMD

for the Atari 800 and Atari 400

March 1981

Version 1.0

TABLE OF CONTENTS

START UP

Editor/Assembler/Debug (EASMD)

FOR START UP:

Put the OSS diskette in disk drive 1 and turn on the power.

This will load the Operating System and execute CP/A. Now enter:

EASMD (return)

This will load the Editor/Assembler/Debug and start executing it. See the CP/A manual for other capabilities.

WARMSTART:

The user can return to CP/A using the EASMD command CP or by using the SYSTEM RESET key. He can then re-enter EASMD by using the CP/A command RUN (if he has not loaded another program). This does a warm start which preserves text lines already in memory.

BACK-UP COPY:

On a dual drive system, simply use COPY or DUPDSK. On a single drive system, one can use DUPDSK or one can make a back-up copy of EASMD on another diskette via the CP/A SAVE command.

System RAM size	32k	40k	48k
Start address	5700	7700	9700
End address	7C00	9C00	BC00
File Name: EASMD.COM	(or any .COM name of your choice)		

NOTE: For a full explanation of CP/A commands see the CP/A reference manual.

SYNTAX CONVENTIONS

The following conventions are used in the discussion of syntax in this manual.

1) Capital letters denote commands, etc. which must be typed by the user exactly as shown. (eg. LIST, DEL)

2) Lower case letters denote types of items which may be used. The various types are shown in the next section. (eg. lno)

3) Items in square brackets are optional (eq. [,lno])

4) Multiple items in braces indicate that any one may be used.
```
(eg. {A} )
     {Q}
```

TYPES OF ITEMS:

The following types of items are used in describing syntax commands.

lno — line number (in range 0 to 65535).

string — A string of ASCII characters.

adr — A memory address (given in hex).

data — A list of hexadecimal values separated by commas.

Example: AB,12,FE

incr — Increment a decimal value.

filespec — See Atari DOS manual or CP/A reference manual for full format.

Generally you may use
```
D[n]:xxxxxxxx.yyy     for disk files
P:                    for the printer
etc.
```
Note that in EASMD filespecs must ALWAYS be prefaced with a pound sign (#).

EDITOR

The Editor allows the user to enter and edit lines of ASCII text.

TEXT FORMAT

Lines of ASCII text received by the Editor are stored in memory. A line consists of a line number (0 to 65535), text information and a carriage return. The text information that is between the line number and the carriage return is stored exactly as it is received. Thus any combination of ASCII data is valid text.

Example:

```
1000LITTLE GREEN APPLES
```

This is valid text as far as the Editor is concerned.

NOTE: The Assembler, however, expects a blank after the line number and will not look at the first character after the line number. Thus

```
1000ABC        LDA     #0
```

is seen as

```
1000 BC        LDA     #0
```

Example:

```
100 PRINT X<SIN(X)
```

The Editor can be used to create and edit Basic programs.

TABLES

The text area and other user tables are built starting at an address in low memory and growing towards high memory. The user can change this address using the LOMEM command.

The user can also change the highest address the Editor will use for user text by using the change memory command in the Debug monitor to change UHIMEM. (See memory map for UHIMEM address).

COMMAND FORMAT

The stored lines of text are manipulated by Editor commands. A command is distinguished from text by the absence of a line number. Any line of data received by the program that does not begin with an ASCII numeric is considered to be a command. The Editor will examine the characters to determine what function to perform. If these characters do not form a valid command, or if the command syntax is invalid, the Editor will respond with:

```
WHAT?
```

LINE PROMPTING

The Editor will prompt the user each time a command has finished executing by printing:

```
                        EDIT
```

The cursor will appear on the following line. Since some commands take awhile to execute, the prompt serves to tell the user when more input is allowed.

EDITOR COMMAND SYNTAX AND DECRIPTION

```
NEW
```

NEW will delete all user text from the text area in memory.

```
DEL     lno
DEL     lno1,lno2
```

DEL deletes the specified line number (lno) or all the lines in the range lno1 through lno2.

```
FIND    /string/
FIND    /string/,A
FIND    /string/lno1[,lno2]
FIND    /string/lno1[,lno2],A
```

The FIND command will search the specified lines (all or lno1 through lno2) for the "string" between the specified delimiters. The delimiters may be any character other than blank. The second delimiter must be the same as the first.

If "A" is specified, any line that contains a matching string will be printed at the user terminal. If "A" is not specified, then only the first line that contains a matching string will be printed.

```
LIST
LIST    #filespec
LIST    lno1[,lno2]
LIST    #filespec,lno1[,lno2]
```

The LIST command will cause all lines in the specified range to be listed to the screen (or to a device/file when "#filespec" is specified).

If "lno1" is less than the line number of the first text line, then listing will start with the first line. If "lno2" is greater than the line number of the last text line, then listing will end with the last line.

Hitting the break key will stop the LIST.

Example: `LIST #D1:EX.TST`

Will list all lines to a file EX.TST on drive 1.

Example: `LIST #P:`

Will list to the printer.

```
PRINT
PRINT    #filespec
PRINT    lno1[,lno2]
PRINT    #filespec,lno1[,lno2]
```

Print is exactly the same as LIST except that the line numbers are not PRINTed, and that the EDIT ready prompt will not be printed after the last line until the user hits the RETURN key.

```
ENTER    #filespec[,M]
```

The ENTER command causes previously LISTed text from the device or file specified by #filespec to be re-entered. The optional "M" parameter specifies that the new text is to be merged with the text currently in memory. If "M" is not present, then the text area will be cleared before starting the ENTER.

Example: ENTER #D2:XXX
                Will re-enter the text that was listed to the file XXX on drive 2.

```
NUM
NUM      slno,incr
NUM      incr
```

The number command is used to automatically attach line numbers to user lines. The user is prompted with the next line number. A blank automatically follows the line number. The "slno" parameter specifies the starting line number. The "incr" parameter is the line number increment.

The default "incr" is 10. The default "slno" is the last text line number plus "incr".

Hitting RETURN after the line number prompt terminates NUMber mode.

```
REN
REN      slno,incr
REN      incr
```

The REN command renumbers the text. The first line number will be "slno". The line numbers will increment by incr. The default "slno" and "incr" is 10.

```
REP      /old string/new string/
REP      /old string/new string/,{A}
                                 {Q}
REP      /old string/new string/lno1[,lno2]
REP      /old string/new string/lno1[,lon2],{A}
                                            {Q}
```

The REP command will search the specified lines (all or lno1 through lno2) for the "old string" (between specified delimiters). The delimiters follow the same rules as the delimiters for FIND.

The "A" option causes all occurrence of "old string" to be replaced with "new string" (between the same specified delimiters).

If the "Q" option is specified then when each match is found, the line is listed and the user is allowed to specify change (Y followed by RETURN) or don't change (RETURN alone) this occurrence. Hitting BREAK will terminate the REPlace and return to the Editor.

If neither "A" or "Q" is specified, only the first occurrence of "old string" will be replaced with "new string".

NOTE: Each time a replace is done the changed line is listed.

SIZE

The SIZE command prints the users low memory address, the highest used memory address, and the highest usable memory address (UHIMEM).

LOMEM adr

LOMEM command changes the address at which user tables start.

NOTE: The LOMEM command will destroy any user statements in memory.

NOTE: This command can be used to reserve a space between the default low memory and the new low memory address. This space can then be used for the object output from the assembler.

CP
DOS

CP or DOS returns to the OSS Control Program (CP/A)

BYE

BYE returns to the Atari Memo Pad.

ASM
ASM [#filespec1], [#filespec2], [#filespec3]

The ASM command assembles source code and produces object code and a listing.

By default:
1) The source "device" is the user text area.
2) The listing "device" is the screen.
3) The object "device" is memory.

These defaults can be overridden as follows:

```
        filespec1 -  source code file or device
        filespec2 -  listing file or device
        filespec3 -  object file or device
```

A "filespec" can be omitted by substituting a comma. in which case the default holds for that parameter.

Example: `ASM #D1:SOURCE,#D2:LIST,#D1:OBJ`

In this example, the source will come from D1:SOURCE, the listing will be written to D2:LIST, and the object will be written to D1:OBJ.

Example: `ASM ,,#D3:OBJ`

In this example the source will come from user text area in memory, the listing will go to the screen, and the object code will be written to the file OBJ on disk drive 3.

Example: `ASM ,#P:`

In this example the listing will go to the printer.

NOTE: See the .OPTion directive for full information about when object is actually written to the specified file (or memory).

BUG

The BUG command causes the debug monitor to be entered.

DEBUG

The Debug Monitor allows the user to perform controlled execution of machine code, examine memory, alter memory, move memory blocks and verify the equality of memory blocks.

COMMAND FORMAT

The Debug Monitor assumes that any line of data that it receives is a command. If the data does not form a valid command, the Debug Monitor responds with:

WHAT?

LINE PROMPTING

The Debug Monitor will signal completion of a command by printing:

DEBUG

The cursor will appear on the following line.

NOTE: If the user is getting a syntax error indication (WHAT?) on what he thinks is a valid command, he should check the prompt message (DEBUG/EDIT) to verify that he is in the correct mode.

DEBUG COMMAND SYNTAX AND DESCRIPTION

G [adr]

The G Command (Go) transfers control to the specified address via a JMP command. If "adr" is not specified, then the current monitor program counter is used.

T [adr]

The T Command (Trace) causes instructions to be executed starting at "adr". If "adr" is not specified then the current monitor program counter is used. As each instruction is executed, its address, mnemonic and operand will be displayed along with the current values in the 6502 A, X, Y, P(status), & S(stack) registers.

Hitting the break key (BREAK) will terminate trace.

S [adr]

The S Command (Step) is exactly like the T command except that only one instruction is executed.

D

D adr1[,adr2]

The D command (Display Memory) will cause memory from "adr1" to "adr2" to be displayed in hexadecimal. If "adr2" is omitted, then 8 bytes are displayed (ie, adr2 = adr1 + 8).
If "adr1" is omitted, then this display starts where the last display left off (ie, at the last "adr2" + 1).

Hitting the break key (BREAK) will terminate Display.

C [adr1]<data

The C command (Change Memory) is used to alter memory starting at "adr". If "adr" is not specified, then Change uses the most recent "adr1" if D was the last command, or the next unchanged address if C was the last command.
The "data" is a list of 1 byte hex values separated by commas.

Example: C 5000<3,CD,1F

Will change locations 5000 thru 5004 to 3,CD,1F,2,3 respectively.

Multiple commas may be used to skip over memory addresses without changing the contents to reach the desired address.

Example: C 5000<3,,1F

will change hex location 5000 to 3, location 5002 to 1F, and location 5001 will be unchanged.

L

L adr1[,adr2]

The L command (list) will cause the instructions located at "adr1" to be disassembled and displayed with the address, instruction mnemonic and operand. If "adr2" is not specified, then twenty instructions will be listed. If the address field ("adr1") is not specified, then this list will start where the last one left off.

Hitting the break key (BREAK) will stop the listing.

M tadr<fsadr,feadr

The M command (Move) moves data from the address "fsadr" through the address "feadr" to the address specified with "tadr".

tadr -	"move to" address
fsadr -	"move from" start address
feadr -	"move from" end address

V   adr1<adr2,adr3

The V Command (Verify) compares the memory starting at "adr1" with the memory located at "adr2" through "adr3". If any of the compared bytes mismatch, then address and data bytes will be displayed.

DR

The DR command (Display Registers) will cause the A,X,Y, status (P) and stack (S) registers to be displayed in hexidecimal.

CR   <data

The CR Command (Change Registers) is used to change the registers. Registers are assumed to be in the order: A,X,Y, status (P) stack (S), so that the first byte of data goes into A register the second into X, etc.

As in the C command, "data" is a list of hexadecimal values separated by commas and field may be skipped by use of multiple commas.

```
Example:        CR<FF,,3

                will set A=FF and Y=3.  It will leave
                X,P and S unchanged.
```

X

The X command (exit) will cause control to return to the Editor.

A

The A command (Assemble) will cause the system to enter into the Debug Assembler mode. No prompt other than the cursor is used in this mode.

The Debug Assembler is a line-at-a-time assembler that uses 6502 mnemonics and operand format. Relative branch operands are specified as the actual "branch to" address; the Assembler creates the relative address.

The format of each line is:

```
[adr]< assembler code
```

The Debug Assembler keeps track of the location counter so that if "adr" is omitted, the next consecutive address is used.

Entering only a carriage return will return the user to the Debug monitor.

Example: While in Debug mode the user enters:

```
A
5000< LDA#3
< BNE $5010
```

The "A" puts the user into the Debug Assembler. The next two statements will cause memory to contain the following:

```
5000  A9 03
5002  D0 0C
```

NOTE: The blank after the "<" is required.

NOTE: The Debug Assembler accepts both decimal and hex numbers as operands; therefore, hex operands must be preceeded by "$".

BREAK POINTS

BRK instructions must be individually set and removed by the user.

Step and Trace intercept the BRK instruction and simulate its execution.

ASSEMBLER

The Assembler gets control when ASM is typed into the Editor. For the ASM command syntax, see the Editor section.

Hitting the break key (BREAK) will stop the assembly.

ASSEMBLER INPUT

Input to the Assembler is lines of ASCII data as entered into the Editor. Source lines are of the form:

```
(line number) (blank) (source statement)
```

where source statement is of the form:

```
[label]          {6502 instruction}
                 {    directive    }
```

A source statement may consist of a label only, or it may be blank.

In general the format is as specified in the MOS Technology 6502 Programming Manual. We recommend that the user unfamiliar with 6502 assembly language programming should purchase:

"Programming the 6502" by Rodney Zaks
or
"6502 Assembly Language Programming" by Lance Leventhal.

INSTRUCTION FORMAT:

A) Instruction mnemonics as described in the MOS Technology 6502 Programming Manual.

B) Immediate operands begin with #

C) "(Operand,X)" and "(Operand),Y" for indirect addressing.

D) "Operand,X" and "Operand,Y" for indexed addressing.

E) Zero page and forward equates recognized and evaluated within the limits of a two pass assembler.

F) "*" refers to the location counter.

G) Comment lines begin with ";"

H) Hex constants begin with "$"

I) The "A" operand is reserved for accumulator addressing.

DIRECTIVES

.TITLE "string"

The .TITLE directive allows the user to specify a title to be used in conjunction with .PAGE

.PAGE ["string"]

The .PAGE directive allows the user to specify a page heading. It issues an ASCII form feed (hex OC) and prints the most recent title and page headings.

NOTE: The most recent title and page headings are also printed every time 52 lines of source code have been assembled.

.BYTE expression and/or "string" list

The .BYTE directive sets a one byte value for each expression and the ASCII equivalent of each character of each string into the object code.

```
Example:      .BYTE  3,"ABC",7,"X"

              produces:

              03 41 42 43 07 58
```

.WORD expresion list

The .WORD directive sets a two byte value into the object code for each expression in the list. The value is in 6502 address order (least significant byte, most significant byte).

```
Example:         .WORD  $1000,$2000

                 produces:

                 00 10 00 20
```

.DBYTE expression list

The .DBYTE directive sets a two bute value into the object code for each expression in the list. The value is in most significant, least significant byte order.

```
Example:         .DBYTE $1000,$2000

                 produces:
```

.TAB expression,expression,expression

The .TAB directive sets displacements for the printing of the op code, operand, and comment fields of the source line. Each expression is a one byte value.
Defaults are 12, 17, 27 .

.OPT assembler option list

The .OPT directive allows the user to specify certain options affecting the assembly.

Possible options are :

LIST/NOLIST
NOOBJ/OBJ
ERR/NOERR
EJECT/NOEJECT

LIST/NOLIST determines if a listing is produced.
NOOBJ/OBJ determines if object code is produced.
ERR/NOERR determines if error messages are printed.
EJECT/NOEJECT determines if a form feed, title, and page are printed after 52 source lines.

Defaults are:

OBJ - when the object is going to a device/file.
NOOBJ - when the object "device" is memory.
LIST, ERR, EJECT - in all cases.

*= expression

The *= directive serves the function of ORG. It sets the current location counter for subsequent source statements.

NOTE: *= must be written with no intervening blanks.

= expression

The = directive is an equate (EQU) statement. It must always be written:

```
LABEL  = expression
```

The value of the "expression" is assigned to "LABEL".

.IF   expression ,label

The .IF statement allows limited conditional assembly.
If the "expression" is true (non-zero), the Assembler skips all following lines up to the one that begins with the "label". If the "expression" is false (zero), assembly continues normally.

NOTE: There can be NO blank between the comma and label.

.INCLUDE   #filespec

The .INCLUDE directive allows source code from the device or file specified in "filespec" to be inserted into the assembly.

NOTE: .INCLUDE's can not be nested. That is, a file that was included cannot contain a .INCLUDE directive.

NOTE: .INCLUDE cannot be the last statement. It must be followed by a .END or some other statement.

.END

The .END directive terminates the assembly.

EXPRESSIONS

Expressions are evaluated strictly left to right. Parentheses are not valid. Valid operators are:

```
        +    -    *    /    &   (& is a binary AND)
```

These are all binary operands. ("-5 + 3" is not valid, but "0 - 5 + 3" is valid.)

```
        Example: LDX # ADDR/256
                 LDY # ADDR&255
                 Will put the MSB and LSB portions of the address
                 of "ADDR" into X and Y respectively.
```

STRINGS:

Strings must be enclosed in double quotes:

```
                              .BYTE  "THIS IS A MESSAGE"
```

The single character representation for the immediate operand :

```
                              #'C
```

LABEL:

Labels must start in the 1st colunm after (line number)(blank). A label may consist of up to 255 characters. It must start with an alpha character and may be followed by alpha-numeric characters or the character "." .

NOTE: The character "A" by itself can not be a label.

COMMENTS:

Comment lines start with the character ";"

No special character is needed to delineate a comment after the assember code on a line. When the assember recognizes the end of the operand field (or op code field for instructions without operands), the rest of the line is assumed to be comment.

NOTE: This can give unexpected results in some cases.

Example:

```
LDA  7A    GET NUM
```

will generate

```
A5 07
```

The decimal number "7" is terminated by the character "A". The comment in this case is:

```
        A     GET NUM
```

If the user wishes to specify the hex location 7A, he must use $7A.

ERROR DESCRIPTION

When an error occurs the system will print out:

```
ERROR- XX [message]
```

Where XX represents an error number. When the Assembler finds more than 1 error in a line, up to 3 error numbers will be listed. Most ERRORs will produce a message (similar to those below).

ERROR NUMBERS

1 - MEMORY FULL

All available memory has been used. If issued from Editor, no more statements can be entered. If issued by the Assembler, no more labels can be defined.

2 - INVALID DELETE RANGE

The first number specified in a delete range does not exist.

3 - DEBUG ASSEMBLER ADDRESS ERROR

The origin address on an input line to the Debug Assembler is incorrectly specified.

4 - BLANK REQUIRED AFTER LINE NUMBER

The Assembler expects the first character after a line number to be a blank. The first character was ignored.

5 - UNDEFINED REFERENCE

Assembler has encountered an undefined label.

6 - ASSEMBLER SYNTAX ERROR

7 - DUPLICATE LABEL

The Assembler has encountered a label that is already defined.

8 - BUFFER OVERFLOW

An internal buffer is full. Try making the source code shorter.

9 - EQUATE HAS NO LABEL

An equate (=) must have a label.

10 - VALUE OF EXPRESSION > 255

The value of an expression was greater than 255 but a one byte value was required.

11 - NULL STRING

A null string is invalid in .BYTE

12 - INVALID ADDRESS OR ADDRESS TYPE

An invalid address type was specified for the mnemonic.

13 - PHASE ERROR

The address generated for a label in pass 2 of the Assembler is different from the address generated by pass 1. Other errors can also cause this error to be generated.

14 - UNDEFINED/FORWARD REFERENCE FOR *= (ORG)

The operand for the *= directive must already be defined when the directive is encountered. A forward reference on an *= directive is invalid.

```
Example:          1000   *=ABC
                  2000 ABC = $1000
              Will produce this error.
```

15 - LINE TOO LONG

The input line is too long. (This error results when there are too many distinct items on a line for the syntax processor to handle.) Break the input line into multiple lines.

16 - INVALID INPUT LINE

The Assembler received a line that does not start with a valid line number.

17 - LINE NUMBER TOO BIG

The line number on an Editor input line is too big. (greater than 65535).

19 - NO ORIGIN (*=) SPECIFIED
Either no origin (*=) was given or it was specified as 0. This error will cause the assembly to terminate.

20 - OVERFLOW ON NUM OR REN

On NUM or REN command the line number generated went over 65535. If REN caused this error, the line numbers are now invalid. Issuing a valid REN command will correct the problem.

21 - NESTED INCLUDE INVALID

An INCLUDEd file can not contain a .INCLUDE directive.

NOTES

LOMEM/HIMEM:

A default low memory address is s[illegible] the system is booted up. EASMD does NOT automatically reset this value.
If a program (for example, a device handler) sets lomem and then EASMD is entered, this address remains unchanged.

EASMD does set a default UHIMEM (highest usable memory for EASMD tables, including user text) which can be changed by using the Change memory command in the Debug monitor.

IOCBs USED:

No command in the Debug monitor does I/O to a device other than the screen or keyboard; therefore, IOCBs 1 through 7 are not used by the system itself while in Debug mode.

Several commands in the Editor however, can do I/O to other devices (ENTER, ASM, etc). In these cases, the Editor must use one or more IOCBs. (The Editor uses IOCBs 1 through 4). Unpredictable things can happen to a file that was allocated to one of these IOCBs and never closed. The user who is debugging code that does I/O needs to be aware of this fact.

ALWAYS CLOSE FILES.

Note that returning to CP/A will ALWAYS cause all files to be closed.

LOAD/SAVE:

To load and save code for debugging, use the CP/A LOAD and SAVE command. To return to EASMD after LOADing a file, the user must enter RUN followed by the coldstart or warmstart address (see memory map). This will work if the user's code did not overlay any memory used by EASMD.

NUMBERS:

The Editor/Assembler/Debug (EASMD) uses positive integers and hex numbers, but it uses a Floating Point package for ASCII to integer conversion. This can give some unexpected results.

```
Example:        LDA    #6.7

                produces

                A9 07

Example:        100.    100.1    99.9

                entered as line numbers each produces
                the line number    100.
```

BASIC:

The Editor can be used to create and edit OSS BASIC A+ programs. Of course, the user must take care of changing line numbers in GOTO, GOSUB, etc. whenever RENumber is used.

MEMORY MAP

The following are some memory addresses used by EASMD which may be of interest to the user. All addresses are given in hex.

size of RAM	[illegible]	40K	48K
zero page free for user	BO-CF	BO-CF	BO-CF
user high memory (UHIMEM)	0498	0498	0498
Coldstart	5700	7700	9700
Warmstart	5703	7703	9703

SYNTAX SUMMARY

EDITOR

```
ASM
ASM       [#source filespec], [#list filespec], [#object filespec]

BUG

BYE

CP

DEL       lno
DEL       lno1,lno2

DOS

ENTER     #filespec

FIND      /string/
FIND      /string/,A
FIND      /string/lno1[,lno2]
FIND      /string/lno1[,lno2],A

LIST
LIST      #filespec
LIST      lno1[,lno2]
LIST      #filespec,lno1[,lno2]

LOMEM     adr

NEW

NUM
NUM       slno,incr
NUM       incr

PRINT
PRINT     #filespec
PRINT     lno1[,lno2]
PRINT     #filespec,lno1[,lno2]

REN       slno,incr
REN       incr

REP       /old string/new string/
REP       /old string/new string/,{A}
                                  {Q}
REP       /old string/new string/lno1[,lno2]
REP       /old string/new string/lno1[,lno2],{A}
                                             {Q}

SIZE
```

DEBUG

A	[adr]< assembler code (blank required after <)
C	[adr1]< data
CR	<data
D	
D	adr1[,adr2]
DR	
G	[adr]
L	
L	adr1[,adr2]
M	tadr < fsadr, feasr
S	[adr]
T	[adr]
V	adr1 < adr2,adr3
X	

ASSEMBLER DIRECTIVES

.BYTE	expression and/or "string" list
.DBYTE	expression list
.END	
.IF	expression,label
.INCLUDE	#filespec
.OPT	option list
.PAGE	["string"]
.TAB	expression, expression, expression
.TITLE	"string"
.WORD	expression list
*=	expression
=	expression

ERROR SUMMARY

This is a summary of error messages produced by the EASMD program. For a more detailed decripition see the section on ERROR DESCRIPTION.

EASMD ERRORS:

1	-	MEMORY FULL
2	-	INVALID DELETE RANGE
3	-	DEBUG ASSEMBLER ADDRESS ERROR
4	-	BLANK REQUIRED AFTER LINE NUMBER
5	-	UNDEFINED REFERENCE
6	-	ASSEMBLER SYNTAX ERROR
7	-	DUPLICATE LABEL
8	-	BUFFER OVERFLOW
9	-	EQUATE HAS NO LABEL
10	-	VALUE OF EXPRESSION > 255
11	-	NULL STRING
12	-	INVALID ADDRESS OR ADDRESS TYPE
13	-	PHASE ERROR
14	-	UNDEFINED/FORWARD REFERENCE FOR *= (ORG)
15	-	LINE TOO LONG
16	-	INVALID INPUT LINE
17	-	LINE NUMBER TOO BIG
19	-	NO ORIGIN (*=) SPECIFIED
20	-	OVERFLOW ON NUM OR REN
21	-	NESTED INCLUDE INVALID

For the user convenience a summary of the error messages that can be generated by DOS and passed to EASMD are included.

DOS ERRORS:

DEC	HEX	MESSAGE
128	(80)	BREAK ABORT
129	(81)	FILE ALREADY OPEN
130	(82)	NON EXISTENT DEVICE
131	(83)	FILE OPENED FOR WRITE ONLY
132	(84)	INVALID COMMAND
133	(85)	DEVICE OR FILE NOT OPEN
134	(86)	INVALID IOCB NUMBER
135	(87)	FILE OPENED FOR READ ONLY
136	(88)	END OF FILE
138	(8A)	DEVICE TIMEOUT
139	(8B)	DEVICE NAK
144	(90)	DEVICE DONE ERROR
146	(92)	FUNCTION NOT IMPLEMENTED
160	(A0)	DRIVE # ERROR
161	(A1)	TOO MANY OPEN FILES (NO SECTOR BUFFER AVAILABLE)
162	(A2)	MEDIUM FULL (NO FREE SECTORS)
163	(A3)	FATAL SYSTEM DATA I/O ERROR
164	(A4)	FILE # MISMATCH
165	(A5)	FILE NAME ERROR
166	(A6)	POINT DATA LENGTH ERROR
167	(A7)	FILE PROTECTED
168	(A8)	COMMAND INVALID (SPECIAL OPERATION CODE)
169	(A9)	DIRECTORY FULL
170	(AA)	FILE NOT FOUND
171	(AB)	POINT INVALID

NOTICE

OSS reserves the right to make changes or improvements in the product described in this manual at any time and without notice.

Optimized Systems Software
10379 Lansdale Avenue
Cupertino, CA 95014
Telephone: 408-446-3099

OPTIMIZED SYSTEMS SOFTWARE

OSS BASIC A+

for the ATARI 800 (R)

MARCH 1981

Version 3.0

CONTENTS

NOTE: Sections Marked with an asterisk (*) are new or substantially changed from standard Atari Basic.

PREFACE

1 GENERAL INFORMATION

2 PROGRAM DEVELOPMENT COMMANDS

3 EDIT FEATURES

6 FUNCTION LIBRARY

7 STRINGS

8 ARRAYS AND MATRICES

9 GRAPHICS MODES AND COMMANDS

10 SOUND AND GAME CONTROLLERS

11 ADVANCED PROGRAMMING TECHNIQUES AND INFORMATION

12 ADVANCED SYSTEM FEATURES

13 PLAYER / MISSILE GRAPHICS

ABOUT THIS MANUAL

This BASIC A+ manual is intended as an "add-on" or appendix to the "BASIC REFERENCE MANUAL" supplied by Atari, Inc. Make sure that your BASIC REFERENCE MANUAL is Atari part number C-015307, REV. 1 !!

GETTING STARTED

To use BASIC A+ with CP/A:

Place the CP/A master disk in drive 1 and turn on the power in the same manner used to boot an Atari disk.

In response to the CP/A prompt "D1:", simply type in "BASIC [return]" and BASIC A+ will load and run.

If you exit from BASIC A+ to CP/A (via DOS or CP commands or via the RESET key), you may return to BASIC A+'s warmstart point by simply entering RUN to CP/A. NOTE: see CP/A manual for circumstances under which this does not work. If necessary, you may use 'RUN addr' from CP/A to enter at BASIC A+'s coldstart or warmstart address. See table below for those addresses.

To use BASIC A+ with Atari's DOS:

Boot an Atari master diskette, and enter the Atari menu DOS.

Put the diskette with BASIC A+ in a disk drive and use the Atari LOAD BINARY FILE from the menu to load BASIC A+.

Use the Atari RUN AT ADDRESS menu command to do a "coldstart" of BASIC A+. The address to use depends upon the amount of free RAM in your system.

If you exit BASIC A+ (via the DOS or CP commands), you may return without losing any program currently in memory by using the Atari menu RUN AT ADDRESS command to do a "warmstart". Again, the warmstart address depends upon the amount of free RAM.

size of free RAM	32k	40k	48k
coldstart address	4400	6400	8400
warmstart address	4403	6403	8403

ERRATA AND MINOR CHANGES

This section contains instructions for making minor changes and insertions to the Atari Basic manual to transform it into a BASIC A+ manual. Some of these changes, however, are necessary because of errors in the Atari manual even as it pertains to Atari Basic.

The changes below include two pages to be inserted at appropriate spots in the manual. The instructions should be self-explanatory, consisting of a location to change and instructions therefor.

CHANGES

Page 2: paragraph headed "variable:"
Change: "...advisable not to use a keyword...", etc.
To: It is perfectly acceptable to use most keywords in or as variable names so long as the assignment explictly uses the word "LET". Some keywords, however, are "poison", including NOT, USING, and STEP.

PAGE 4: Paragraph headed "Logical Expression"
Note: Logical expressions are a subset of arithmetic expressions. Thus,

```
        LET A=(B<C)
```

is legal, as "B<C" is a logical (and thus arithmetic) expression.

Page 6: Arithmetic Operators
Delete: First line (The ATARI....)
Add: BASIC A+ uses 7 arithmetic operators:

& Bitwise "and" of the positive integers (both<=65535)
! Bitwise "or" of two positive intergers

Pages 9 & 10 CONT and LET

Replace descriptions of these statements with those on next two pages following, which may be inserted in manual as pages 9-a and 10-a.

---

footnote: pages marked as this one is, "--information page only--", are not part of the final combined manual but are simpy instructions for putting the manual together.

CHANGES, CONTINUED

Page 10: NEW
Change: "Used in Direct Mode."
To: Normally used in Direct Mode, but useful in deffered mode as an alternative to END

Page 14: [SHIFT] [DELETE]
Add: Caution: does not delete BASIC program lines!

Page 15: FOR
Add: Note: see also SET/SYS() discussion in chapter 12.

Page 18: IF/THEN
Add: See also IF...ELSE...ENDIF discussion in BASIC A+ appendix to this chapter.

Page 19: Thrid paragraph
Change: "The statements R=9:GOTO 100...."
To: "The statements R=9:GOTO 200...."

Page 22: TRAP
Add: Note: see also CONT (page 9) and ERR() [in BASIC A+ appendix to chapter 6].

Page 22: Last Line
Change: 32767 to 32768
Add to sentence: or whose value is zero(0).

Page 23: 4th paragraph
Change: "BASIC reserves IOCB #0..."
To: BASIC A+ uses IOCB #0 for I/O to the screen editor, and the user may take advantage of this fact by using GET #0,A or PRINT #0;... or using #0 with virtually any I/O statements. The user may even CLOSE #0 but should do so with EXTREME caution.

Page 25: DOS
Add: CP [as a keyword title]
Add Note CP is identical in function to DOS.

Page 25: INPUT
Add: Note: In BASIC A+, input variables may be subscripted, with results similar to LET.

Page 25: INPUT
Add at bottom of page: If the user's sole response to an INPUT prompt is [CONTROL-C][return], a special error (number 27) will be issued by INPUT. This can be useful in data entry manipulations.

Page 28: PRINT
Add: Note: See also PRINT USING in BASIC A+ appendix to this chapter.

Page 29: Second paragraph
Delete: paragraph
Add: String and matrix variables used in READ statements must be dimensioned and MAY be subscripted.

NOTE: String DATA may be enclosed in quotes, in which case commas may be contained in the string data.

Page 29: STATUS
Delete: Entire description of statement
Add: The STATUS statement places the current static status of the specified file into the specified variable (avar). The "Device Status Routine" is NOT called, so the value may or may not reflect the true current dynamic status. Use XIO to access dynamic status.

Page 30: XIO
Add: Note: It is highly recommended that the BASIC user avoid XIO cmdno's 3,5,7,9,11,17,37 and 38. BASIC A+ users should find all these, as well as cmdno's 32 thru 36, totally unnecessary.

Page 30: cmdno 13 of XIO
Change: Example of command 13 ("same as Basic")
To: Should be followed by BASIC A+ status

Page 30: description of aexp1 and aexp2
Change: "control bytes"
To: control words

Page 31: Modifying a BASIC program on disk
Change step 5: "READY"
TO: OSS CP/A prompt.
Add step 5a: Load BASIC A+ by typing BASIC [return].

Page 36: USR
Add: Note: See also SET/SYS() in chapter 12.

Page 39: Fourth paragraph
Change: "...a substring contains up to 99 characters..."
To: Any string or substring may contain up to 32767 characters (depending upon available memory).

Page 39: Figure 7.5
Note: In BASIC A+, lines 50 and 60 may be replaced by:

```
50 A$=A$,B$,C$
```

Page 39: Under "String Splitting"
Add To: Beginning of sentence which begins "The starting location cannot..."
Add: For source strings only (i.e, strings used in an expression)...
Note: Destination strings [in A$=...,READ A$(X),INPUT A$(10,20)] have no subscript restrictions other than their dimension.

Page 42: Second paragraph ("Note ...")
Delete: Paragraph and following sample program
Add: Note: BASIC A+ always initializes arrays AND strings when they are DIMensioned. Array elements are set to all nulls (binary zeros).

Page 42: Figure 8-4
Note: Lines 30 and 40 may be replaced by
30 READ A(E)

Page 63: PROGRAMMING IN MACHINE LANGUAGE
Add Note to second paragraph: See also SET/SYS() in CH 12.

Page D-2 (appendix D): FREE RAM
Note: BASIC A+ gives the user more zero page free RAM than Atari Basic, but uses more RAM in page 600.
Change: FREE RAM addresses to read:

1791	6FF	FREE RAM
1664	680	
	...	
207	CF	FREE BASIC A+ and EASMD RAM
192	C0	
191	BF	FREE EASMD RAM
176	B0	

Page I-1 (Appendix I): STOPLN
Delete Line: STOPLN not supported
Reason: Use ERR(1) instead.

Page I-1: ERRSAV
Delete Line: ERRSAV not supported
Reason: Use ERR(0) instead.

Page I-1: PTABW
Delete Line: PTABW not supported
Reason: Use SET 1,xx instead.

Page 117: Index
Note: Index has not yet been updated to reflect additions of BASIC A+ features. Also, page number (117) is not correct.
Help: Send in your software registration form to get on our FREE newsletter mailing list. We will NOT send newsletter to anyone not returning this form.

--information page only--

CHAPTER APPENDICES

The following pages are intended to be appendices to the various chapters of the Atari Basic manual. As such, they have page numbers that should make it obvious where they are to be inserted in the manual. For example, 12-A and 12-B are to be inserted after page 12 (chapter 2) in the manual.

Please read these pages thoroughly, as much of the most important material of the BASIC A+ manual is contained herein.

--information page only--

CONT

(CON.)

```
Format:          CONT
Example:         CONT
                 100 CONT
```

In direct mode, this command resumes a program after a STOP statement or BREAK key abort or any stop caused by an error.

Caution: Execution resumes on the line following the halt. Statements on the same line as and following a STOP or error will not be executed.

In deferred mode, CONT may be used for error trap handling.

```
Example:         10 TRAP 100
                 20 OPEN #1,12,0,"D:X"
                 30
                 ..
                 ..
                 100 IF ERR(0)=170 THEN
                     OPEN #1,8,0,"D:X":CONT
```

In line 20 we attempt to open a file for updating. If the file does not exist, a trap to line 100 occurs. If the "FILE NOT FOUND" error occured, the file is opened for output (and thus created) and execution continues at line 30 via "CONT".

LET

```
Format:         [LET] avar=aexp
                [LET] svar=sexp[,sexp...]
Exapmle:        LET X=3.5
                LET LETTER$="a"
                A$="*",A$,A$,A$,A$,A$
```

Normally an optional keyword, LET must be used to assign a value to a variable name which starts with (or is identical to) a reserved name.

String concattenation may be accomplished via the for shown in the last example above . Note that a concatenation of the form

```
        A$=B$,C$
```

is exactly equivalent to

```
        A$=B$
        A$(LEN(A$)+1)=C$
```

```
Examples:       DIM A$(100),B$(100)
                A$="123"
                B$="ABC"
                A$=A$,B$,A$

        (At this point, A$= "123ABC123ABC")

                A$(4,9)="X",STR$(3*7),"X"

        (At this point, A$="123X21X23ABC")

                A$(7)=A$(1,3)

        (Finally, A$="123X21123")
```

TRACE

TRACEOFF

Formats:	TRACE TRACEOFF
Examples:	100 TRACE TRACEOFF

These statements are used to enable or disable the line number trace facility of BASIC A+. When in TRACE mode, the line number of a line about to be executed is displayed on the screen surrounded by square brackets.

Exceptions: The first line of a program does not have its number traced. The object line of a GOTO or GOSUB and the looping line of FOR or WHILE may not be traced.

Note: A direct statement (e.g., RUN) is TRACED as having line number 32768.

LVAR

Format:	LVAR filename
Example:	LVAR "E:"

This statement will list (to any file) all variables currently in use. The example will list the variables to the screen. Strings are denoted by a trailing '$', arrays by a trailing '('.

LOMEM

Format:	LOMEM addr
Example:	LOMEM DPEEK(128)+1024

This command is used to reserve space below the user's program space. The user then might use the space for assembly language routines. The usefulness of this may be limited, though, since there are other more usable reserved areas available.

Caution: LOMEM wipes out any user program currently in memory.

DEL

```
Format:          DEL line[,line]
Example:         DEL 1000,1999
```

DEL deletes program lines currently in memory. If two line numbers are given (as in the example), all lines between the two numbers (inclusive) are deleted. A single line number deletes a single line.

Example:

```
        100  DEL 1000,1999
        110  SET 9,1:TRAP 1000
        120  ENTER "D:OVERLAY1"
        1000 REM THESE LINES ARE DELETED BY
        1010 REM LINE 100
        1020 REM
        1030 REM PRESUMABLY THEY WILL BE
        1040 REM OVERLAID BY THE ENTERED PROGRAM
        1990 REM SEE 'ENTER' AND 'SET' FOR
        1999 REM MORE INFO
```

ADVANCED PROGRAM CONTROL

BASIC A+ adds Structured Programming capability with two new Program Control Structures.

IF...ELSE...ENDIF

```
Format:         IF aexp: <statements>
                [ELSE: <statements> ]
                 ENDIF
Examples:       200 IF A>100:PRINT "TOO BIG"
                210 A=100
                220 ELSE:PRINT "A-OK"
                230 ENDIF


                1000 IF A>C : B=A : ELSE : B=C : ENDIF
```

BASIC A+ makes available an exceptionally powerful conditional capability via IF...ELSE...ENDIF

In the format given, if the expression evaluates non-zero then all statements between the following colon and the corresponding ELSE (if it exists) or ENDIF (if no ELSE exists) are executed; if ELSE exists, the statements between it and ENDIF are skipped.

If the aexp evaluates to zero, then the statements (if any) between the colon and ELSE are skipped and those between ELSE and ENDIF are executed. If no ELSE exists, all statements through the ENDIF are skipped.

The colon following the aexp IS REQUIRED and MUST be followed by a statement. The word THEN is NOT ALLOWED in this format

There may be any number (including zero) of statements and lines between the colon and the ELSE and between the ELSE and the ENDIF.

The second example above sets B to the larger of the values of A and C.

Note: IF structures may be nested.

Example:

```
        100 if A>B : REM SO FAR A IS BIGGER
        110   IF A>C : PRINT "A BIGGEST"
        120   ELSE : PRINT "C BIGGEST"
        130   ENDIF
        140 ELSE
        150   IF B>C : PRINT "B BIGGEST"
        160   ELSE : PRINT "C BIGGEST"
        170   ENDIF
        180 ENDIF
```

WHILE

ENDWHILE

Format: WHILE aexp : <statements> : ENDWHILE

Example:

```
100 A=3
110 WHILE A: PRINT A
120   A=A-1 : ENDWHILE
```

With WHILE, the BASIC A+ user has yet another powerful control structure available. So long as the aexp of WHILE remains non-zero, all statements between WHILE and ENDWHILE are executed.

Example:

```
WHILE 1 : ....
```

The loop executes forever

Example:

```
WHILE 0 : ....
```

The loop will never execute

Caution: Do not GOTO out of a WHILE loop or a nesting error will likely result. (though POP can fix the stack in emergencies.)

Note: The aexp is only tested at the top of each passage through the loop.

Note: As with ALL BASIC A+ control structures, WHILEs may be nested as deep as memory space allows.

INPUT

```
Format:          INPUT string-literal,var[,var..]
Example:         INPUT "3 VALUES >>",V(1),V(2),V(3)
```

BASIC A+ allows the user to include a prompt with the INPUT statement to produce easier to write and read code. The literal prompt ALWAYS replaces the default ("?") prompt. The literal string may be nul for no prompt at all.

Note: No file number may be used when the literal prompt is present.

Note: In the example above, if the user typed in only a single value followed by RETURN, he would be reprompted by BASIC A+ with "??". But see chapter 12 for variations available via SET.

DIR

```
Format:          DIR filespec
Example:         DIR "D:*.COM"
```

List the contents of a directory to the screen. Action is similar to CP/A DIR command, but there are no default file specifications. The example above would list all COMmand files on drive 1.

PROTECT

UNPROTECT

```
Format:          PROTECT filespec
                 UNPROTECT filespec
Examples:        PROTECT "D:*.COM"
                 100 UNPROTECT "D2:JUNK.BAS
```

PROTECTing a file implies that the file cannot be erased or written to. UNPROTECT eliminates any existing protection. Similar to CP/A PROtect and UNProtect, but there are no default file specifications. In the examples, the first would protect all command files on drive 1 and the second would unprotect only the file shown.

ERASE

```
Format:          ERASE filespec
Example:         ERASE "D:*.BAK
```

Erase will erase any unprotected files which match the given filespec. The example would erase all .BAK (back-up) files on drive 1. Similar to CP/A ERAse, but there are no default file specifiers.

RENAME

```
Format:          RENAME <filespec,filename>
Example:         RENAME "D2:NEW.DAT,OLD.BAK"
```

Allows renaming file(s) from BASIC A+. Note that the comma shown MUST be imbedded in the string literal or variable used as the file parameter.

Caution: It is strongly suggested that wild cards (* and ?) NOT be used when RENAMing.

PRINT USING

```
Format:          PRINT   [#fn; ]USING sexp, exp [, exp...]
Example:         (see below)
```

PRINT USING allows the user to specify a format for the output to the device or file associated with "fn" (or to the screen). The format string "sexp" contains one or more format fields. Each format field tells how an expression from the expression list is to be printed. Valid format field characters are:

& * + - $, . % ! /

Non-format characters terminate a format field and are printed as they appear.

```
Example 1) 100 PRINT USING "## ###X#",12,315,7

        2) 100 DIM A$(10) : A$="## ###X#"
           200 PRINT USING A$,12,315,7
```

Both 1) and 2) will print

```
12 315X7
```

Where a blank separates the first two numbers and an X separates the last two.

NUMERIC FORMATS:

The format characters for numeric format fields are:

& * + - $, .

DIGITS (# & *)

Digits are represented by:

& *

- Indicates fill with leading blanks
& - Indicates fill with leading zeroes
* - Indicated fill with leading asterisks

If the number of digits in the expression is less than the number of digits specified in the format then the digits are right justified in the field and preceded with the proper fill character.

NOTE: In all the following examples b is used to represent a blank.

Example:

Value	Format Field	Print Out

1	###	bb1
12	###	b12
123	###	123
1234	###	234
12	&&&	012
12	***	*12

DECIMAL POINT(.)

A decimal point in the format field indicates that a decimal point be printed at that location in the number. All digit positions that follow the decimal point are filled with digits. If the expression contains fewer fractional digits than are indicated in the format, then zeroes are printed in the extra positions. If the expression contains more fractional digits than indicated in the format, then the expression is rounded so that the number of fractional digits is equal to the number of format positions specified.

A second decimal point is treated as a non-format character.

Example:

Value	Format Field	Print Out
123.456	###.##	123.46
4.7	###.##	bb4.70
12.35	##.##.	12.35.

COMMA (,)

A comma in the format field indicates that a comma be printed at that location in the number. If the format specifies a comma be printed at a position that is preceeded only by fill characters (0 b *) then the appropriate fill character will be printed instead of the comma.

The comma is a valid format character only to the left of the decimal point. When a comma appears to the right of a decimal point, it becomes a non-format character. It terminates the format field and is printed like a non-format character.

Example:

Value	Format Field	Print Out
5216	##,###	b5,216
3	##,###	bbbbb3
4175	**,***	*4,175
3	&&,&&&	000003
42.71	##.##,	42.71,

SIGNS (+ -)

A plus sign in a format field indicates that the sign of the number is to be printed. A minus sign indicates that a minus sign is to be printed if the number is negative and a blank

Example:

Value	Format Field	Print Out
34.2	$$$$$.##	bb$34.20
34.2	+$$$$$.##	+bb$34.20
1572563.41	$$,$$$,$$$.##+	$1,572,563.41+

NOTE: There can only be one floating character per format field.

NOTE: +, - or $ in other than proper positions will give strange results.

STRING FORMATS:

The format characters for string format fields are:

% - Indicates the string is to be right justified.
! - indicates the string is to be left justified.

If there are more characters in the string than in the format field, than the string is truncated.

Example:

Value	Format Field	Print Out
ABC	%%%%	bABC
ABC	!!!!	ABCb
ABC	%%	AB
ABC	!!	AB

ESCAPE CHARACTER (/)

The escape character (/) does not terminate the format field but will cause the next character to be printed, thus allowing the user to insert a character in the middle of the printing of a number.

Example:

```
PRINT USING "###/-####",2551472          prints

               255-1472
```

Example:

```
100 AREA = 408
200 NUM = 2551472
300 PHONE = (AREA*1E+7)+NUM
400 DIM F$(20)
500 F$ = "(###/)###/-####"
600 PRINT USING F$,PHONE
700 END
```

This program will print

```
          (408)255-1472
```

NOTE: Improperly specified format fields can give some very strange results.

NOTE: The function of "," and ";" in PRINT are overridden in the expression list of PRINT USING, but when file number "fn" is given then the following "," or ";" have the same meaning as in PRINT. So to avoid an initial tabbing, use a semicolon (;).

Example:

```
PRINT #5; USING A$,B
```

Will print B in the format specified by A$ to the file or device associated with file number 5.

Example:

```
PRINT USING "## /* #=###",12,5,5*12
```

```
12 * 5=60
```

Example:

```
PRINT USING "TOTAL=##.#+",72.68
```

```
TOTAL=72.7+
```

Example:

```
100 DIM A$(10) : A$="TOTAL="
200 DIM F$(10) : F$="!!!!!!##.#+"
300 PRINT USING F$,A$,72.68
```

```
TOTAL=72.7+
```

NOTE: IF there are more expressions in the expression list than there are format fields, the format fields will be reused.

Example:

```
PRINT USING "XX##",25,19,7
```

will print

```
XX25XX19XXb7
```

WARNING:

A format string must contain at least one format field. If the format string contains only non-format characters, those characters will be printed repeatedly in the search for a format field.

TAB

Format: TAB [#fn,] aexp

Example: TAB #PRINTER,20

TAB outputs spaces to the device or file specified by fn (or the screen) up to column number "aexp". The first column is column 0.

NOTE: The column count is kept for each device and is reset to zero each time a carriage return is output to that device. The count is kept in AUX2 of the IOCB. (See OS documemtation).

NOTE: If "aexp" is less than the current column count, a carriage return is output and then spaces are put out up to column "aexp".

BPUT

Format: BPUT #fn, aexp1, aexp2
Example: (see below)

BPUT outputs a block of data to the device or file specified by "fn". The block of data starts at address "aexp1" for a length of "aexp2".

NOTE: The address may be a memory address. For example, the whole screen might be saved. Or the address may be the address of a string obtained using the ADR function.

Example:

```
BPUT #5, ADR(A$), LEN(A$)
```

This statements writes the block of data contained in the string A$ to the file or device associated with file number 5.

BGET

Format: BGET #fn, aexp1, aexp2
Example: (see below)

BGET gets "aexp2" bytes from the device or file specified by "fn" and stores them at address "aexp1".

NOTE: The address may be a memory address. For example, a screen full of data could be displayed in this manner. Or the address may be the address of a string. In this case BGET does not change the length of the string. This is the user's responsibility.

Example:

```
10 DIM A$(1025)
20 BGET #5,ADR(A$),1024
30 A$(1025) = CHR$(0)
```

This program segment will get 1024 bytes from the file or device associated with file number 5 and store it in A$. Statement 30 sets the length of A$ to 1025.

NOTE: No error checking is done on the address or length so care must be taken when using this statement.

RPUT

Format: RPUT #fn, exp [,exp...]
Example: (see below)

RPUT allows the user to output fixed length records to the device or file associated with "fn". Each "exp" creates an element in the record.

NOTE: A numeric element consists of one byte which indicates a numeric type element and 6 bytes of numeric data in floating point format.

A string element consists of one byte which indicates a string type element 2 bytes of string length, 2 bytes of DIMensioned length, and then X bytes where X is the DIMensioned length of the string.

Example:

```
100 DIM A$(6)
200 A$ = "XY"
300 RPUT #3,B,A$,10
```

Puts 3 elements to the device or file asscoiated with file number 3. The first element is numeric (the value of B). The second element is a string (A$) and the third is a numeric (10). The record will be 26 bytes long, (7 bytes for each numeric, 5 bytes for the string header and 6 bytes (the DIM length) of string data).

RGET

Format:

```
RGET     #fn, {svar} [,{svar}...]
              {avar} [,{avar}...]
```

Example: (see below)

RGET allows the user to retreive fixed length records from the device or file associated with file number "fn" and assign the values to string or numeric variables.

NOTE: The type of the element in the file must match the type of the variable (ie. they must both be strings or both be numeric).

Example:

```
1) RPUT #5,A
2) RGET #1,A$
```

If 1) is a statement in a program used to generate a file and 2) is a statement in another program used to read the same file, an error will result.

NOTE: When the type of element is string, then the DIMensioned length of the element in the file must be equal to the DIMensioned length of the string variable.

Example:

```
1) 100 DIM A$(100)
      .
      .
   800 RPUT #3,A$
      .
      .
```

```
2) 100 DIM X$(200)
      .
      .
   800 RGET #2,X$
      .
      .
```

If 1) is a section of a program used to write a file and 2) is a section of another program used to read the same file, then an error will occur as a result of the difference in DIM values.

NOTE: RGET sets the correct length for a string variable (the length of a string variable becomes the actual length of the string that was RPUT - not necessarily the DIM length).

Example:

```
1)100 DIM A$(10)
  200 A$ = "ABCDE"
   .
   .
  800 RPUT #4,A$

2)100 DIM X$(10)
  200 X$ = "HI"
   .
   .
  800 RGET #6,X$
  900 PRINT LEN(X$),X$
   .
   .
```

If 1) is a section of a program used to create a file and 2) is a section of another program used to read the file then it will print:

```
5       ABCDE
```

DPEEK

DPOKE

```
Format:         DPEEK(addr)
                DPOKE addr,aexp
Examples:       PRINT "variable name table is at";DPEEK(130)
                DPOKE 741,DPEEK(741)-1024
```

The DPEEK function and DPOKE statement parallel PEEK and POKE. The difference is that, instead of working with single byte memory locations, DPEEK and DPOKE access or change Double byte locations (or "words"). Hence, DPEEK may return a value from 0 to 65535; and DPOKE's aexp may be any expression evaluating to a like range.

The primary advantage of DPEEK over DPOKE is illustrated by the following two exactly equivalent program fragments:

```
100 A=PEEK(130)+256*PEEK(131)
100 A=DPEEK(130)
```

In the second example at the head of this section, the top of memory is lowered by 1k bytes in a single, easy-to-read statement.

ERR

```
Format:         ERR(aexp)
Example:        PRINT "ERROR";ERR(0); "OCCURRED AT LINE";ERR(1)
```

This function--in conjunction with TRAP, CONT, and GOTO allows the BASIC A+ programmer to effectively diagnose and dispatch virtually any run-time error.

ERR(0) returns the last run-time error number
ERR(1) returns the line number where the error occurred

Example:

```
100 TRAP 200
110 INPUT "A NUMBER, PLEASE >>",NUM
120 PRINT "A VALID NUMBER" : END
200 IF ERR(0)=8 THEN GOTO ERR(1)
210 PRINT "UNEXPECTED ERROR #";ERR(0)
```

· TAB

Format: TAB(aexp)
Example: PRINT #3; "columns:"; TAB(20); 20; TAB(30); 30

The TAB function's effect is identical with that of the TAB statement (page 32-A+). The difference is that, for PRINT statements, an imbedded TAB function simplifies the programmers task greatly (see the example).

TAB will output ATASCII space characters to the current PRINT file or device (#3 in our example). Sufficient spaces will be output so that the next item will print in the column specified (only if TAB is followed by a semi-colon, though). If the column specified is less than the current column, a RETURN will be output first.

Caution: The TAB function will output spaces on some device whenever it is used; therefore, it should be used ONLY in PRINT statements. It will NOT function properly in PRINT USING.

ADVANCED STRINGS

SUBSTRINGS:

A destination string is one that is being assigned to. Any other string is a source string. In

```
READ X$
INPUT X$
X$=Y$
```

X$ is the destination string, Y$ is the source string.

Substrings are defined as follows:

STRING	definition when destination string	definition when source string
S$	the entire string 1 thru DIM value	from 1st thru LEN character
S$(n)	from nth thru DIMth character	from nth thru LENgth character
S$(n,m)	from the nth thru the mth character	from the nth thru the mth character

It is an error if either the first or last specified character (n and m, above) is outside the DIMensioned size. It is an error if the last character position given (explicitly or implicitly) is less than the first character position.

```
Example:        Assume: DIM A$(10)
                        A$ = "VWXYZ"

                1) PRINT A$(2)       prints:
                   WXYZ

                2) PRINT A$(3,4)     prints:
                   XY

                3) PRINT A$(5,5)     prints:
                   Z

                4) PRINT A$(7)
                   is an error because A$ has a length of 5.
```

NOTE: Refer to the LET statement, page 10-a, for examples of BASIC A+ string concatenation.

FIND

```
Format:          FIND(sexp1,sexp2,aexp)
Example:         PRINT FIND ("ABCDXXXXABC","BC",N)
```

FIND is an efficient, speedy way of determining whether any given substring is contained in any given master string.

FIND will search sexp1, starting at position aexp, for sexp2. If sexp2 is found, the function returns the position where it was found, relative to the beginning of sexp1. If sexp2 is not found, a 0 is returned.

In the example above, the following values would be PRINTed:

```
        2 if N=0 or N=1
        9 if N>2 and N<10
        0 if N>=10
```

More Examples:

```
        10 DIM A$(1)
        20 PRINT "INPUT A SINGLE LETTER:
        30 PRINT "Change/Erase/List"
        40 INPUT "CHOICE ?",A$
        50 ON FIND("CEL",A$,0) GOTO 100,200,300
```

An easy way to have a vector from a menu choice

```
        100 DIM A$(10): A$="ABCDEFGHIJ"
        110 PRINT FIND (A$,"E",3)
        120 PRINT FIND (A$(3),"E")
```

Line 110 will print "5" while 120 will print "3". Remember, the position returned is relative to the start of the specified string.

```
        100 INPUT "20 CHARACTERS, PLEASE:",A$
        110 ST=0
        120 F=FIND(A$,"A",ST):IF F=0 THEN STOP
        130 IF A$(F+1,F+1)="B" OR A$(F+1,F+1)="C"
                THEN ST=F+1:GOTO 120
        140 PRINT "FOUND 'AB' OR 'AC'"
```

This illustrates the importance of the aexp's use as a starting position.

ADVANCED GAME CONTROL

Note: See also chapter 13, PLAYER/MISSILE GRAPHICS.

HSTICK

VSTICK

```
Formats:          HSTICK(aexp)
                  VSTICK(aexp)
EXAMPLES:         IF HSTICK(0)>0 and VSTICK(0)<0
                     THEN PRINT "DOWN, TO THE RIGHT"
```

If the numbering scheme for STICK(0) positions dismayed you, take heart: HSTICK and VSTICK provide a simpler method of reading the joysticks.

```
VSTICK(n) reads joystick n and returns:
      +1 if the joystick is pushed up
      -1 if the joystick is pushed down
       0 if the joystick is vertically centered

HSTICK(n) reads joystick n and returns:
      +1 if the joystick is pushed right
      -1 if the joystick is pushed left
       0 if the joystick is horizontally centered
```

PEN

```
Format:           PEN(aexp)
Example:          PRINT "light pen at X=";pen(0)
```

The PEN function simply reads the ATARI light pen registers and returns their contents to the user.

```
PEN(0) reads the horizontal position register
PEN(1) reads the vertical position register
```

NUMBERS

All numbers in Basic are in BCD floating point.

RANGE:

Floating point numbers must be less than 10E+98 and greater than or equal to -10E-98.

INTERNAL FORMAT:

Numbers are represented internally in 6 bytes. There is a 5 byte mantissa containing 10 BCD digits and a one byte exponent.

The most significant bit of the exponent byte gives the sign of the mantissa (0 for postive, 1 for negative). The least significant 7 bits of the exponent byte gives the exponent in excess 64 notation. Internally, the exponent represents powers of 100 (not powers of 10).

```
        Example:        0.02 = 2 * 10^-2 = 2 * 100^-1

        exponent=       -1 + 40 = 3F

           0.02 =       3F 02 00 00 00 00
```

The implied decimal point is always to the right of the first byte. An exponent less than hex 40 indicates a number less than 1. An exponent greater than or equal to hex 40 represents a number greater than or equal to 1.

Zero is represented by a zero mantissa and a zero exponent.

In general, numbers have a 9 digit precision. For example, only the first 9 digits are significant when INPUTing a number. Internally the user can usually get 10 significant digits in the special case where there are an even number of digits to the right of the decimal point (0,2,4...).

ADDITIONAL CHAPTERS

The pages that follow constitute two new chapters to be added to the Atari Basic manual in the process of turning it into a BASIC A+ manual.

Chapter 12 describes some of the system features that give the BASIC A+ programmer even more control over the functions and presumptions of the language. Using some of the features described in chapter 12 can get you in real trouble...or can give you power never before possible in virtually any Basic.

Chapter 13 is almost a manual in and to itself: it explores the world of Player/Missile Graphics, formerly accessible only through poorly documented PEEKs and POKEs and/or slow Basic programs.
The speed and scope of Player/Missile Graphics is probably one of the Atari's most advanced features...and now YOU, the BASIC A+ user, can have almost total control.

--instruction page only--

12

ADVANCED SYSTEM FEATURES

SET and SYS

Formats: SET aexp1,aexp2
SYS(aexp)
Examples: SET 1,5
PRINT SYS(2)

SET is a statement which allows the user to exerices control over a varity of BASIC A+ system level functions. SYS is simply an arithmetic function used to check the SETtings of these functions. The table below summarizes the various SET table parameters. (Default values are given in parentheses.)

aexp1 PARAMETER #		aexp2 LEGAL VALUES	meaning
0,	(0)	0	-BREAK key functions normally
		1	-User hitting BREAK cause an error to occur (TRAPable)
		128	-BREAKs are ignored
1,	(10)	1 thru 12	-Tab "stop" setting fort the comma in PRINT statements.
2,	(63)	0 thru 255	-Prompt character for INPUT (default is "?").
3,	(0)	0	-FOR...NEXT loops always execute at least once (ala ATARI BASIC).
		1	-FOR loops may execute zero times (ANSI standard)
4,		0	-On a mutiple variable INPUT, if the user enters too few items, he is reprompted (e.g. with "??")
	(1)	1	-Instead of reprompting, a TRAPable error occurs.
5,		0	-Lower case and inverse video characters remain unchanged and can cause syntax errors.
	(1)	1	-For program entry ONLY, lower case letters are converted to upper case and inverse video characters are uninverted. Exception: characters between quotes remain unchanged.

6,	(0)	0	-Print error messages along with error numbers (for most errors)
		1	-Print only error numbers.
7,	(0)	0	-Missiles (in Player/Missile-Graphics), which move vertically to the edge of the screen, roll off the edge and are lost.
		1	-Missiles wraparound from top to bottom and vise versa.
8,		0	-Don't push (PHA) the number of parameters to a USR call on the stack [advantage: some assembly language subroutines not expecting parameters may be called by a simple USR(addr)].
	(1)	1	-DO push the count of parameters (ATARI BASIC standard).
9,	(0)	0	-ENTER statements return to the READY prompt level on completion
		1	-If a TRAP is properly set, ENTER will execute a GOTO the TRAP line on end-of-entered-file.

Note: The SET parameters are reset to the system defaults on execution of a NEW statement.

Note: System defaults may be changed either temporarily or permanently (by SAVEing a patched BASIC A+ via CP/A) by POKEing the locations noted in the memory map.

Examples:

```
1) SET 1,4 : PRINT 1,2,3,4
```

THe number will be printed every four columns

```
2) SET 2,ASC(">")
```

Changes the INPUT prompt from "?" to ">"

```
3) 100 SET 9,1 : TRAP 120
   110 ENTER "D:OVERLAY.LIS"
   120 REM execution continues here after entry of
   130 rem the overlay

4) 100 SET 0,1 : TRAP 200
   110 PRINT "HIT BREAK TO CONTINUE"
   120 GOTO 110
   200 REM come here via BREAK KEY

5) 100 SET 3,1
   110 FOR I = 1 TO 0
   120 PRINT " THIS LINE WON'T BE EXECUTED"
   130 NEXT I
```

MOVE

Format:	MOVE from-addr, to-addr, len [MOVE aexp, aexp, aexp]
Example:	MOVE 13*4096, 8*4096, 1024
Caution:	Be careful with this command.

MOVE is a general purpose byte move utility which will move any number of bytes from any address to any address at assembly language speed. NO ADDRESS CHECKS ARE MADE!!

The sign of the third aexp (the length) determines the order in which the bytes are moved.

```
    If the length is postive:
        (from) -> (to)
        (from+1) -> (to+1)
          ...       ...
        (from+len-1) -> (to +len-1)

    If the length is negative:
        (from+len-1) -> (to+len-1)
        (from+len-2) -> (to+len-2)
           ...            ...
        (from+1) -> (to +1)
        (from) -> (to)
```

The example above will move the character set map to BASIC A+'s reserved area in a 48K RAM system (it moves from $D000 to $8000).

PLAYER / MISSILE GRAPHICS

This section describes the BASIC A+ commands and functions used to access the Atari's Player-Missile Graphics. Player Missile Graphics (hereafter usually referred to as simply "PMG") represent a portion of the Atari hardware totally ignored by Atari Basic and Atari OS. Even the screen handler (the "S:" device) knows nothing about PMG. BASIC A+ goes a long way toward remedying these omissions by adding six (6) PMG commands (statements) and two (2) PMG functions to the already comprehensive Atari graphics. In addition, four other statements and two functions have significant uses in PMG and will be discussed in this section.

The PMG statements and functions:

PMGRAPHICS PMCOLOR PMCLR
PMMOVE PMWIDTH MISSILE
BUMP(...) PMADR(...)

The related function and statements:

MOVE BGET BPUT
POKE USR(...) PEEK(...)

AN OVERVIEW

For a complete technical discussion of PMG, and to learn of even more PMG "tricks" than are included in BASIC A+, read the Atari document entitled "Atari 400/800 Hardware Manual" (Atari part number C016555, Rev. 1 or later).

It was stated above that the "S:" device driver knows nothing of PMG, and in a sense this is proper: the hardware mechanisms that implement PMG are, for virtually all purposes, completely separate and distinct from the "playfield" graphics supported by "S:". For example, the size, position, and color of players on the video screen are completely independent of the GRAPHICS mode currently selected and any COLOR or SETCOLOR commands currently active. In Atari (and now BASIC A+) parlance, a "player" is simply a contiguous group of memory cells displayed as a vertical stripe on the screen. Sounds dull? Consider: each player (there are four) may be "painted" in any of the 128 colors available on the Atari (see Setcolor for specific colors). Within the vertical stripe, each bit set to 1 paints the player's color in the corresponding pixel, while each bit set to 0 paints no color at all! That is, any 0 bit in a player stripe has no effect on the underlying playfield display.

Why a vertical stripe? Refer to Figure PMG-1 for a rough idea of the player concept. If we define a shape within the bounds of this stripe (by changing some of the player's bits to 1's), we may then move the stripe anywhere horizontally by a simple register POKE (or via the PMMOVE command in BASIC A+). We may move the player vertically by simply doing a circular shift on the contiguous memory block representing the player (again, the PMMOVE command of BASIC A+ simplifies this process). To simplify:

A player is actually seen as a stripe on the screen 8 pixels wide by 128 (or 256, see below) pixels high. Within this stripe, the user may POKE or MOVE bytes to establish what is essentially a tall, skinny picture (though much of the picture may consist of 0 bits, in which case the background "shows through"). Using PMMOVE, the programmer may then move this player to any horizontal or vertical location on the screen. To complicate:

For each of the four players there is a corresponding "missile" available. Missiles are exactly like players except that (1) they are only 2 bits wide, and all four missiles share a single block of memory, (2) each 2 bit sub-stripe has an independent horizontal position, and (3) a missile always has the same color as its parent player. Again, by using the BASIC A+ commands (MISSILE and PMMOVE, for example), the programmer/user need not be too aware of the mechanisms of PMG.

CONVENTIONS

1. Players are numbered from 0 through 3. Each player has a corresponding missile whose number is 4 greater then that of its parent player, thus missiles are numbered 4 through 7. In the BUMP function, the "playfields" are numbered from 8 through 11, corresponding to actual playfields 0 through 3. (Note: playfields are actually COLORs on the main GRaphics screen, and can be PLOTted, PRINTed, etc).

2. There is some inconsistency in which way is "UP". PLOT, DRAWTO, POKE, MOVE, etc are aware that 0,0 is the top left of the screen and that vertical position numbering increases as you go down the screen. PMMOVE and VSTICK, however, do only relative screen positioning, and define "+" to be UP and "-" to be DOWN. [If this really bothers you please let us know!].

3. "pmnum" is an abbreviation for Player-Missile NUMber and must be a number from 0 to 3 (for players) or 4 to 7 (for missiles).

FIGURE PMG-1

Graphic Representation of Player/Missile Displays vs. Playfield

FIGURE PMG-2

Memory Usage in Player/Missile Graphics

NOTE: assumes 48K system. Adjust addresses downward
8K or 16K for 40k or 32K systems.

Resolution:	single line	double line
Top of RAM	$C000	$C000
Player 3	$BFFF $BF00	$BFFF $BF80
Player 2	$BEFF $BE00	$BF7F $BF00
Player 1	$BDFF $BD00	$BEFF $BE80
Player 0	$BCFF $BC00	$BE7F $BE00
Missiles (all)	$BBFF $BB00	$BDFF $BD80

PMGRAPHICS
(PMG.)

Format:	PMGRAPHICS aexp
Example:	PMG. 2

This statement is used to enable or disable the Player-Missile Graphics system. The aexp should evaluate to 0, 1, or 2:

PMG.0	Turn off PMG
PMG.1	Enable PMG, single line resolution
PMG.2	Enable PMG, double line resolution

Single and Double line resolution (hereafter refered to as "PMG Modes") refer to the height which a byte in the player "stripe" occupies - either one or two television scan lines. (A scan line height is the pixel height in GRaphics mode 8. GRaphics 7 has pixels 2 scan lines high, similar to PMG.2)

The secondary implication of single line versus double line resolution is that single line resolution requires twice as much memory as double line, 256 bytes per player versus 128 bytes. Figure PMG-2 shows PMG memory usage in BASIC A+, but the user really need not be aware of the mechanics if the PMADR function is used.

PMCLR

Format:	PMCLR pmnum
Example:	PMCLR 4

This statement "clears" a player or missile area to all zero bytes, thus "erasing" the player/missile. PMCLR is aware of what PMG mode is active and clears only the appropriate amounts of memory. CAUTION: PMCLR 4 through PMCLR 7 all produce the same action -- ALL missiles are cleared, not just the one specified. To clear a single missile, try the following:

```
SET 7,0 : PMMOVE 4;255
```

PMCOLOR
(PMCO.)

```
Format:           PMCOLOR pmnum,aexp,aexp
Example:          PMCOLOR 2,13,8
```

PMCOLORs are identical in usage to those of the SETCOLOR statement except that a player/missile set has its color chosen. Note there is no correspondence in PMG to the COLOR statement of playfield GRaphics: none is necessary since each player has its own color.

The example above would set player 2 and missile 6 to a medium (luminace 8) green (hue 13).

NOTE: PMG has NO default colors set on power-up or SYSTEM RESET.

PMWIDTH
(PMW.)

```
Format:           PMWIDTH pmnum,aexp
Example:          PMWIDTH 1,2
```

Just as PMGRAPHICs can select single or double pixel heights, PMWIDTH allows the user to specify the screen width of players and missiles. But where PMGRAPHICs selects resolution mode for all players and missiles, PMWIDTH allows each player AND missile to be separately specified. The aexp used for the width should have values of 1,2, or 4 -- representing the number of color clocks (equivalent to a pixel width in GRaphics mode 7) which each bit in a player definition will occupy.

NOTE: PMG.2 and PMWIDTH 1 combine to allow each bit of a player definition to be equivalent to a GRaphics mode 7 pixel -- a not altogether accidental occurence.

NOTE: Although players may be made wider with PMWIDTH, the resolution then suffers. Wider "players" made be made by placing two or more separate players side-by-side.

PMMOVE

```
Format:          PMMOVE pmnum[,aexp][;aexp]
Example:         PMMOVE 0,120;1
                 PMMOVE 1,80
                 PMMOVE 4;-3
```

Once a player or missile has been "defined" (via POKE, MOVE, GET, or MISSILE), the truly unique features of PMG under BASIC A+ may be utilized. With PMMOVE, the user may position the player/missile shape anywhere on the screen almost instantly.

BASIC A+ allows the user to position each player and missile independently. Because of the hardware implementation, though, there is a difference in how horizonal and vertical positioning are specified.

The parameter following the comma in PMMOVE is taken to be the ABSOLUTE position of the left edge of the "stripe" to be displayed. This position ranges from 0 to 255, though the lowest and highest positions in this range are beyond the edges of the display screen. Note the specification of the LEFT edge: changing a player's width (see PMWIDTH) will not change the position of its left edge, but will expand the player to the right.

The parameter following the semicolon in PMMOVE is a RELATIVE vertical movement specifier. Recall that a "stripe" of player is 128 or 256 bytes of memory. Vertical movement must be accomplished by actual movement of the bytes within the stripe - either towards higher memory (down the screen) or lower memory (up the screen). BASIC A+ allows the user to specify a vertical movement of from -255 (down 255 pixels) to +255 (up 255 pixels).

NOTE: The +/- convention on vertical movement conforms to the value returned by VSTICK.

```
Example:        PMMOVE N;VSTICK(N)
```

Will move player N up or down (or not move him) in accordance with the joystick position.

NOTE: SET may be used to tell PMMOVE whether an object should "wraparound" (from bottom of screen to top of screen or vice versa) or should disappear as it scrolls too far up or down. SET 7,1 specifies wraparound. SET 7,0 disables wraparound.

MISSILE

(MIS.)

Format: MISSILE pmnum,aexp,aexp
Example: MISSILE 4,48,3

The MISSILE statement allows an easy way for a parent player to "shoot" a missile. The first aexp specifies the absolute vertical position of the beginning of the missile (0 is the top of screen), and the second aexp specifies the vertical height of the missile.

Example: MISSILE 4,64,3

Would place a missile 3 or 6 scan lines high (depends on PMG. mode) at pixel 64 from the top.

NOTE: MISSILE does NOT simply turn on the bits corresponding to the position specified. Instead, the bits specified are exclusive-or'ed with the current missile memory. This can allow the user to erase existing missiles while creating others.

Example:
```
MISSILE 5,40,4
MISSILE 5,40,8
```

The first statement creates a 4 pixel missile at vertical position 20. The second statement erases the first missile and creates a 4 pixel missile at vertical position 24.

PMG FUNCTIONS

PMADR

```
Format:         PMADR(aexp)
Example:        P0=PMADR(0)
```

This function may be used in any arithmetic expression and is used to obtain the memory address of any player or missile. It is useful when the programmer wishes to MOVE,POKE,BGET, etc. data to (or from) a player area. See next section on "PMG RELATED STATEMENTS" for examples and hints.

NOTE: PMADR(m) -- where m is a missile number (4 through 7) returns the same address for all missiles.

BUMP

```
Format:         BUMP(pmnum,aexp)
Examples:       IF BUMP(4,1) THEN ...
                B=BUMP(0,8)
```

BUMP is a function which can be used in any arithmetic expression. BUMP accesses the collision registers of the ATARI and returns a 1 (collision occured) or 0 (no collision occured) as appropriate for the pair of objects specified. Note that the second parameter (the aexp) may be either a player number or playfield number (8 through 11).

Valid BUMPs:	PLAYER to PLAYER (0-3 to 0-3)
	MISSILE to PLAYER (4-7 to 0-3)
	PLAYER to PLAYFIELD (0-3 to 8-11)
	MISSILE to PLAYFIELD (4-7 to 8-11)

NOTE: BUMP (p,p), where the p's are 0 through 3 and identical, always returns 0.

NOTE: It is advisable to reset the collision registers if a relatively long time has occurred since they were last checked. A dummy usage of BUMP [e.g., JUNK=BUMP(0,0)] will clear the registers.

NOTE

See also decriptions of these statements in preceding sections. The discussions here pertain only to their usage with PMG.

POKE and PEEK

One of the most common ways to put player data into a player stripe may well be to use POKE. In conjunction with PMADR, it is easy to write understandable player loading routines.

Example:

```
100 FOR LOC=48 TO 52
110 READ N: POKE LOC+PMADR(0),N
120 NEXT LOC
...
900 DATA 255,129,255,129,255
```

PEEK might be used to find out what data is in a particular player location.

MOVE

MOVE is an efficient way to load a large player and/or move a player vertically by a large amount. With its ability to MOVE data in upwards or downwards movement, interesting overlap possibilities occur. Also, it would be easy to have several player shapes contained in stripes and then MOVEd into place at will.

Examples:

```
MOVE ADR(A$),PMADR(2),128
```

could move an entire double line resolution player from A$ to player stripe number 2.

```
POKE PMADR(1),255
MOVE PMADR(1),PMADR(1)+1,127
```

would fill player 1's stripe with all "on" bits, creating a solid stripe on the screen.

FIGURE PMG-1

Graphic Representation of Player/Missile Displays vs. Playfield

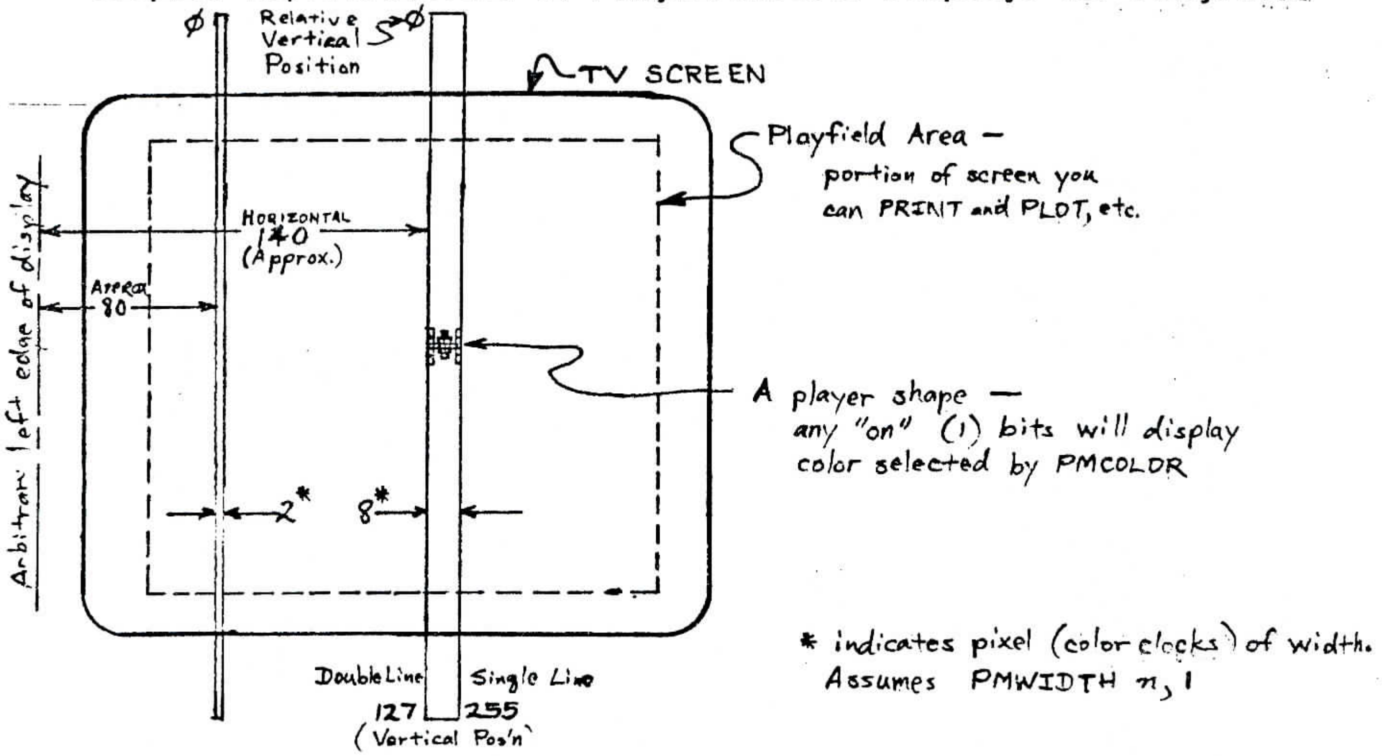

Ø
Relative Vertical Position
Ø
TV SCREEN
Playfield Area — portion of screen you can PRINT and PLOT, etc.
Arbitrary left edge of display
HORIZONTAL 140 (Approx.)
Area 80
A player shape — any "on" (1) bits will display color selected by PMCOLOR
2*
8*
Double Line
Single Line
127
255
(Vertical Pos'n)
* indicates pixel (color clocks) of width. Assumes PMWIDTH n, 1

FIGURE PMG-2

Memory Usage in Player/Missile Graphics

NOTE: assumes 48K system. Adjust addresses downward 8K or 16K for 40k or 32K systems.

Resolution:	single line	double line
Top of RAM	$C000	$C000
Player 3	$BFFF $BF00	$BFFF $BF80
Player 2	$BEFF $BE00	$BF7F $BF00
Player 1	$BDFF $BD00	$BEFF $BE80
Player 0	$BCFF $BC00	$BE7F $BE00
Missiles (all)	$BBFF $BB00	$BDFF $BD80

BGET and BPUT

As with MOVE, BGET may be used to fill a player memory quickly with a player shape. The difference is that BGET may obtain a player directly from the disk!

```
Example:         BGET #3,PMADR(0),128
```

Would get a PMG.2 mode player from the file opened in slot #3.

```
Example:         BGET #4,PMADR(4),256*5
```

Would fill all the missiles AND players in PMG.1 mode -- with a single statement!

BPUT would probably be most commonly used during program development to SAVE a player shape (or shapes) to a file for later retrieval by BGET.

USR

Because of USR's ability to pass parameters to an assembly language routine, complex PMG functions (written in assembly language) can be easly interfaced to BASIC A+.

```
Example:         A=USR(PMBLINK,PMADR(2),128)
```

Might call an assembly language program (at address PMBLINK) to BLINK player 2, whose size is 128 bytes.

EXAMPLE PMG PROGRAMS

1. A very simple program with one player and its missile

```
100 setcolor 2,0,0       : rem note we leave ourselves in GR.0
110 PMGRAPHICS 2         : rem double line resolution
120 let width=1 : y=48 : rem just initializing
130 PMCLR 0 : PMCLR 4    : rem clear player 0 and missile 0
135 PMCOLOR 0,13,8       : rem a nice green player
140 p=PMADR(0)           : rem gets address of player
150 for i=p+y to p+y+4 : rem a 5 element player to be defined
160   read val           : rem see below for DATA scheme
170   poke i,val         : rem actually setting up player shape
180   next i
200 for x=1 to 120       : rem player movement loop
210   PMMOVE 0,x         : rem moves player horizontally
220   sound 0,x+x,0,15 : rem just to make some noise
230   next x
240 MISSILE 0,y,1        : rem a one-high missile at top of player
250 MISSILE 0,y+2,1      : rem another, in middle of player
260 MISSILE 0,y+4,1      : rem and again at top of player
300 for x=127 to 255     : rem the missile movement loop
310   PMMOVE 4,x         : rem moves missile 0
320   sound 0,255-x,10,15
330   IF (x & 7) = 7     : rem every eighth horizontal position
340     MISSILE 0,y,5    : rem you have to see this to believe it
350     ENDIF            : rem could have had an ELSE, of course
360   next x
370 PMMOVE 0,0           : rem so width doesn't change on screen
400 width=width*2        : rem we will make the player wider
410 if width > 4 then width = 1 : rem until it gets too wide
420 PMWIDTH 0,width      : rem the new width
430 PMCLR 4              : rem no more missile
440 goto 200             : rem and do all this again
500 rem THE DATA FOR PLAYER SHAPE
510 data  153            : rem $99          *  **  *
520 data  189            : rem $BD          * **** *
530 data  255            : rem $FF          ********
540 data  189            : rem $BD          * **** *
550 data  153            : rem $99          *  **  *
```

CAUTION : do NOT put the REMarks on lines 510 thru 550 !!!!!!!
(DATA must be last statement on a line !)

Notice how the data for the player shape is built up...
draw a picture on an 8-wide by n-high piece of grid paper, filling in whole cells. Call a filled in cell a '1' bit, empty cells are '0'. Convert the 1's and 0's to hex notation and thence to decimal.

This program will run noticably faster if you use multiple statements per line. It was written as above for clarity, only.

2. A more complicated program, sparsely commented.

```
100 dim hex$(15),t$(4) : hex$="123456789ABCDEF"
110 graphics 0         : rem not necessary, just prettier
120 PMGRAPHICS 2 : PMCLR 0 : PMCLR 1
130 setcolor 2,0,0 : PMCOLOR 0,12,8 : PMCOLOR 1,12,8
140 p0 = PMADR(0) : p1 = PMADR(1) : rem addr's for 2 players
150 v0 = 60 : vold = v0  :rem starting vertical position
160 h0 = 110           : rem  starting horizontal position
200 for loc =v0-8 to v0+7 : rem a 16-high double player
210  read t$           : rem a hex string to t$
220  poke p0+loc,16*FIND(hex$,t$(1,1),0) + FIND(hex$,t$(2,2),0)
230  poke p1+loc,16*FIND(hex$,t$(3,3),0) + FIND(hex$,t$(4,4),0)
       : rem we find a hex digit in the hex string; its decimal
             value is its position (becuz if digit is zero it is
             not found so FIND returns 0 ! )
240  next loc
300 rem  ANIMATE IT
310 let radius=40 : deg  : rem 'let' required, RAD is keyword
320 WHILE 1            : rem forever !!!
330   c=int(16*rnd(0)) : pmcolor 0,C,8 : pmcolor 1,C,8
340   for angle = 0 to 355 step 5  : rem in degrees, remember
350     vnew = int( v0 + radius * sin(angle) )
360     vchange = vnew - vold  : rem change in vertical position
370     hnew = h0 + radius * cos(angle)
380     PMMOVE 0,hnew;vchange : PMMOVE 1,hnew+8;vchange
               : rem move two players together
390     vold = vnew
400     sound 0,hnew,10,12 : sound 1,vnew,10,12
410     next angle
420   rem just did a full circle
430 ENDWHILE
440 rem we better NEVER get to here !

500 rem the fancy data !         8421842184218421
510 DATA 03C0                   !      ****      !
520 DATA 0C30                   !    **    **    !
530 DATA 1008                   !   *        *   !
540 DATA 2004                   !  *          *  !
550 DATA 4002                   ! *            * !
560 DATA 4E72                   ! *  ***  ***  * !
570 DATA 8A51                   !*   * *  * *   *!
580 DATA 8E71                   !*   ***  ***   *!
590 DATA 8001                   !*              *!
600 DATA 9009                   !*  *        *  *!
610 DATA 4812                   ! *  *      *  * !
620 DATA 47E2                   ! *   ******   * !
630 DATA 2004                   !  *          *  !
640 DATA 1008                   !   *        *   !
650 DATA 0C30                   !    **    **    !
660 DATA 03C0                   !      ****      !
```

Notice how much easier it is to use the hex data. With FIND, the hex to decimal conversion is easy, too.

The factor slowing this program the most is the SIN and COS being calculated in the movement loop. If these values were pre-calculated and placed in an array this program would move!

EXTENDED ERROR DESCRIPTIONS

The error number explanations in the Atari Basic manual, while adequate, sometimes fail to give all possible reasons that a user might get zapped with one. For this reason, and because BASIC A+ has added several new error messages of its own, we have included a new set of Error Descriptions.

Note that I/O related explanations are not included. The best source of explanations for I/O errors is probably the Atari Dos Manual.

Note that the messages printed by BASIC A+ are shown at the top of each description (beside the error number).

--information page only--

ERROR NUMBER DECRIPTION

1 - BREAK KEY ABORT

While SET 0,1 was specified, the operator hit the BREAK key. This trappable error gives the BASIC A+ programmer total system control.

2 - MEM FULL

All avaiable memory has been used. No more statements can be entered and no more variables (arithmetic, string or array) can be defined.

3 - VALUE

An expression or variable evaluates to an incorrect value.

Example: An expression that can be converted to a two byte integer in the range 0 to 65235 (hex FFFF) is called for and the given expression is either too large or negative.

```
A = PEEK(-1)
DIM B(70000)
```

Both these statments will produce a value error

Example: An expression that can be converted to a one byte integer in the range 0 to 255 hex(FF) is called for and the given expression is too large.

```
POKE 5000,750
```

This statement produces a value error.

Example: `A=SQR(-4)` Produces a value error.

4 - TOO MANY VARS

No more variables can be defined. The maximum number of variables is 128.

5 - STRING LEN

A character beyond the DIMensioned or current length of a string has been accessed.

Example:

```
1000 DIM A$(3)
2000 A$(5) = "A"
```

This will produce a string length error at line 2000 when the program is RUN.

6 - READ, NO DATA

A READ statement is executed but we are already at the end of the last DATA statement.

7 - LINE #/VAL > 32767

A line number larger than 32767 was entered.

8 - INPUT/READ

The INPUT or READ statement did not recieve the type of data it expected.

Example:

```
INPUT A
```

If the data entered is 12AB then this error will result.

Example:

```
1000 READ A
2000 PRINT A
3000 END
4000 DATA 12AB
```

Running this program will produce this error.

9 - DIM

Example: A string or an array was used before it was DIMensioned.

Example: A previously DIMensioned string or array is DIMensioned again.

```
1000 DIM A(10)
2000 DIM A(10)
```

This program produces a DIM error.

10 - EXPR TOO COMPLEX

An expression is too complex for Basic to handle. The solution is to break the calculation into two or more Basic statements.

11 - OVERFLOW

The floating point routines have produced a number that is either too large or too small.

12 - NO SUCH LINE #

The line number required for a GOTO or GOSUB does not exist.
The GOTO may be implied as in:

```
1000 IF A=B THEN 500
```

The GOTO/GOSUB may be part of an ON statement.

13 - NEXT, NO FOR

A NEXT was encountered but there is no information about a FOR with the same variable.

Example:

```
1000 DIM A(10)
2000 REM FILL THE ARRAY
3000 FOR I = 0 TO 10
4000 A(I) = I
5000 NEXT I
6000 REM PRINT THE ARRAY
7000 FOR K = 0 TO 10
8000 PRINT A(K)
9000 NEXT I
10000 END
```

Running this program will cause the following output:

```
0

ERROR-  13 AT LINE 9000
```

NOTE: Improper use of POP could cause this error.

14 - LINE TOO LONG

The line just entered is longer than Basic can handle. The solution is to break the line into multiple lines by putting fewer statements on a line, or by evaluating the expression in multiple statements.

15 - LINE DELETED

The line containing a GOSUB or FOR was deleted after it was executed but before the RETURN or NEXT was executed.
This can happen if, while running a program, a STOP is executed after the GOSUB or FOR, then the line containing the GOSUB or FOR is deleted, then the user types CONT and the program tries to execute the RETURN or NEXT.

Example:

```
1000 GOSUB 2000
1100 PRINT "RETURNED FROM SUB"
1200 END
2000 PRINT "GOT TO SUB"
2100 STOP
2200 RETURN
```

If this program is run the print out is:

```
GOT TO SUB

STOPPED    AT LINE 2100
```

Now if the user deletes line 1000 and then types CONT we get

```
ERROR- 15 AT LINE 2200
```

16 - RETURN, NO GOSUB

A RETURN was encountered but we have no information about a GOSUB.

Example:

```
1000 PRINT "THIS IS A TEST"
2000 RETURN
```

If this program is run the print out is:

```
THIS IS A TEST

ERROR- 16 AT LINE 2000
```

NOTE: improper use of POP could also cause this error.

17 - BAD LINE

If when entering a program line a syntax error occurs, the line is saved with an indication that it is in error. If the program is run without this line being corrected, execution of the line will cause this error.

NOTE: The saving of a line that contains a syntax error can be useful when LISTing and ENTERing programs.

18 - NOT NUMERIC

If when executing the VAL function, the string argument does not start with a number, this message number is generated.

Example: `A = VAL("ABC")` produces this error.

19 - LOAD, TOO BIG

The program that the user is trying to LOAD is larger than available memory.

This could happen if the user had used LOMEM to change the address at which Basic tables start, or if he is LOADing on a machine with less memory than the one on which the program was SAVEd.

20 - FILE #

If the device/file number given in an I/O statement is greater than 7 or less than 0, then this error is issued.

Example: `GET #8,A`

will produce this error.

21 - NOT SAVE FILE

This error results if the user tries to LOAD a file that was not created by SAVE.

22 - 'USING' FORMAT

This error occurs if the length of the entire format string in a PRINT USING statement is greater than 255. It also occurs if the length of the sub-format for one specific variable is greater than or equal to 60.

23 - 'USING' TOO BIG

The value of a variable in a PRINT USING statement is greater than or equal to 1E+50.

24 - 'USING' TYPE

In a PRINT USING statement, the format indicates that a variable is a numeric when in fact the variable is a string. Or the format indicates the variable is a string when it is actually a numeric.

Example:

```
PRINT USING "###",A$
PRINT USING "%%%",A
```

Will produce this error.

25 - DIM MISMATCH

The string being retreived by RGET from a device (ie. the one written by RPUT) has a different DIMension length than the string variable to which it is to be assigned.

26 - TYPE MISMATCH

The record being retreived by RGET (ie. the one written by RPUT) is a numeric, but the variable to which it is to be assigned is a string. Or the record is a string, but the variable is a numeric.

27 - INPUT ABORT

An INPUT statement was executed and the user entered cntl-C (return).

28 - NESTING

The end of a control structure such as ENDIF or ENDWHILE was encountered but the run-time stack did not have the corresponding beginning structure on the Top of Stack.

Example:

```
        10 While 1 : Rem loop forever
        20 gosub 100
        100 ENDWHILE
```

Endwhile finds the GOSUB on Top of Stack and issues the error.

29 - PLAYER/MISSILE NUMBER

Players must be numbered from 0-3 and missiles from 4-7.

30 - PM GRAPHICS NOT ACTIVE

The user attempted to use a PMG statement other than PMGRAPHICS before executing PMGRAPHICS 1 or PMGRAPHICS 2.

31 - FATAL SYSTEM ERROR

Record circumstances leading to this error and report it to us immediately.

32 - END OF 'ENTER'

This is the error resulting from a program segment such as:

```
        SET 9,1 : TRAP line# : ENTER filename
```

when the ENTER terminates normally.

NEW APPENDICES

The following pages intended to be three new Appendices to the Atari Basic manual, again with the purpose of properly upgrading it to a BASIC A+ manual.

READ APPENDIX J CAREFULLY !

Appendix J lists the known points of incompatibility between standard Atari Basic and BASIC A+. You will be surprised to find how minor the differences are (and how easy it is to get around even these differences).

Appendix K is our attempt to provide you with a usable index. It lists all keywords AS WELL AS THE STATEMENT SYNTAX associated with them and gives a page number reference. We hope you find it useful.

Appendix L will be useful to those of you who wish to customize BASIC A+ in some way.

--information page only--

APPENDIX J

COMPATIBILITIES

The following incompatibilities between Atari Basic and BASIC A+ are known to exist:

1. BASIC A+ and Atari Basic SAVEd program files are NOT COMPATIBLE !!! However, the LISTed form of all Atari Basic programs IS compatible with BASIC A+.
   Solution: use Atari cartridge to LOAD all SAVEd programs, then LIST these programs to a diskette, then go to BASIC A+ and ENTER them and (optional) then SAVE them in BASIC A+ form.

2. Various documented RAM locations do not agree. The only three locations known to be of any significance are now deemed to be too volatile to document. Instead, alternative methods of accessing their purposes are provided:
   STOPLN -- contained line # where a program stopped or found an error -- NOW accessible via ERR(1).
   ERRSAV -- contained the last run-time error number -- NOW accessible via ERR(0).
   PTABW -- the 'tab' size used by PRINT when 'tabbing' for a comma -- NOW accessible via SET 1,<ptabw>.

3. By default, BASIC A+ allows the user to enter program text in lower case, inverse video, or upper case characters. Atari Basic allowed only upper case (non-inverse video) characters. Normally, this is not a problem; however, REMarks and DATA statements ENTERed which contain inverse video and/or lower case characters will find that these characters have been changed to normal video, upper case. Reason: BASIC A+ changes all inverse or lower case character strings NOT ENCLOSED IN QUOTES.
   Solutions:
   a. Put quotes into REMarks and DATA statements as needed.
   b. SET 5,0 -- this will disable entering of lower case and inverse characters; but if you are ENTERing an Atari Basic program, there will be none of these anyway.

4. This one is really exotic: When using XIO, the two parameters normally set to zero (XIO cmd,#file,0,0,FL$) represent BYTES with Atari Basic. With BASIC A+, they represent WORDS (double bytes). Reason: in Atari CIO, each IOCB has six (6) "AUX" bytes. With Atari Basic, the 2 parameters were placed in AUX3 and AUX4. With BASIC A+, the parameters are placed in AUX3-AUX4 (first word parameter) and AUX5-AUX6 (second word). Obviously, this allows much more data to be passed to some device drivers that may actually use the AUX bytes sometime. NOTE: there are no known current Atari drivers that use

these bytes at all, so unless you have custom drivers the difference is unnoticable.

5. Similarly exotic: When OPENing a file, there is a (usually) dummy parameter normally set to zero (as in OPEN #file,mode,0,FL$). As with XIO, this parameter, AS WELL AS THE MODE parameter, represent BYTE values in Atari Basic. With BASIC A+, both parameters are WORD values. In Atari Basic, the mode is placed in AUX1 and the second parameter in AUX2. In BASIC A+, the mode is placed in AUX1-AUX2 and the second parameter in AUX3-AUX4. Again, this was changed to allow more exotic device drivers to receive more information from Basic.
NOTE: there are no known simple situations that use AUX2 through AUX4, so the situation may be moot to you. Some exotic S: (screen) capabilities, though, may be accessible via AUX2. If you ever run into such a situation, follow this example:

```
          Atari:     OPEN #file,mode,special,FILE$
          BASIC A+:  OPEN #file,mode+256*special,0,FILE$
```

Again, this is an unlikely situation to have occur. The BASIC A+ method was chosen because of its compatibility with some Apple II capabilities.

6. ATARI vs. APPLE II: If you are a software author, there are obvious advantages in having one BASIC A+ which will run programs unchanged on two machines. Excepting for GRaphics capabilities, Player/Missile Graphics, SOUND, and some game controls, BASIC A+ is completely compatible on the two machines. Even graphics are compatible to some degree, but see the Apple II BASIC A+ manual for more details.

7. Cartridge convenience: If you did not purchase CP/A (why not?) BASIC A+ may seem a little awkward to use, what with having to LOAD it via the DOS menu, etc. Partial solution: after duplicating the OSS master disk, RENAME the file BASIC.COM to AUTORUN.SYS on any Atari DOS version 2S or 2.8 master disk. Then, when you turn on the power, DOS will boot and immediately run BASIC A+. Of course, you must still use RUN AT ADDRESS to return to BASIC A+ after going to DOS, but you should need to do that less frequently now that BASIC A+ gives you so many extended DOS-like commands. Good luck. And try CP/A soon -- remember it INCLUDES (at NO extra charge) an Editor/ Assembler/Debug package upward compatible with Atari's cartridge (sound familiar ?) .

APPENDIX K

SYNTAX SUMMARY AND KEYWORD INDEX

All keywords, grouped by statements and then functions, are listed below in alphabetical order. A page number reference is given to enable the user to quickly find more information about each keyword.

STATEMENTS

page	syntax
32-H	*BGET #fn, addr, len
32-H	*BPUT #fn, addr, len
9	BYE
24	CLOAD
26	CLOSE #fn
43	CLR
48	COLOR aexp
9	CONT
25	*CP
24	CSAVE
28	DATA <ascii data>
35	DEG
12-B	*DEL line [,line]
41	DIM svar(aexp)
41	DIM mvar(aexp[,aexp])
32-A	*DIR filename
25	DOS
36-A	*DPOKE addr,aexp
48	DRAWTO aexp,aexp
22-A	*ELSE {see IF}
9	END
22-A	*ENDIF {see IF}
22-B	*ENDWHILE
25	ENTER filename
32-B	*ERASE filename
15	FOR avar=aexp TO aexp [STEP aexp]
28	GET #fn, avar
16	GOSUB line
17	GOTO line
45	GRAPHICS aexp
18	IF aexp THEN <stmts>
18	IF aexp THEN line

22-A	*IF aexp : <stmts>
	ELSE : <stmts>
	ENDIF

32-A	*INPUT "...",var [,var...]
25	INPUT [#fn,] var [,var...]
10-A	*[LET] svar=sexp [,sexp..]
10-A	[LET] avar=aexp
10-A	[LET] mvar=aexp

10	LIST [filename]
10	LIST [filename,] line [,line]
26	LOAD filename
48	LOCATE aexp,aexp,avar
12-A	*LOMEM addr
26	LPRINT [exp [;exp...] [,exp...]]
12-A	*LVAR filename
78	*MISSILE pm,aexp,aexp
71	*MOVE fromaddr,toaddr,lenaexp
10	NEW
15	NEXT avar
26	NOTE #fn, avar,avar
20	ON aexp GOTO line [,line...]
20	ON aexp GOSUB line [,line...]
26	OPEN #fn, mode,avar,filename
49	PLOT aexp,aexp
75	*PMCLR pm
76	*PMCOLOR pm,aexp,aexp
75	*PMGRAPHICS aexp
77	*PMMOVE pm[,aexp] [;aexp]
76	*PMWIDTH pm,aexp
28	POINT #fn, avar,avar
35	POKE addr,aexp
20	POP
49	POSITION aexp,aexp
28	PRINT [#fn]
28	PRINT exp [[;exp...] [,exp...]] [;]
28	PRINT #fn [[;exp...] [,exp...]] [;]
32-C	*PRINT [#fn,] USING sexp , [exp[,exp...]]
32-B	*PROTECT filename
28	PUT #fn, aexp
35	RAD
28	READ var [,var...]
10	REM <any remark>
32-B	*RENAME filenames
21	RESTORE [line]
16	RETURN
32-I	*RGET #fn, asvar [,asvar...]
32-H	*RPUT #fn,exp[,exp...]
11	RUN [filename]
29	SAVE filename
69	*SET aexp,aexp
50	SETCOLOR aexp,aexp,aexp
57	SOUND aexp,aexp,aexp,aexp
29	STATUS #fn, avar
15	STEP {see FOR}
11	STOP
32-G	*TAB [#fn], avar
18	THEN {see IF}
15	TO {see FOR}
12-A	*TRACE
12-A	*TRACEOFF
22	TRAP line
32-B	*UNPROTECT filename
22-B	*WHILE aexp
30	XIO aexp,#fn,aexp,aexp,filename
28,32-C	? {same as PRINT}

FUNCTIONS

page	syntax
33	ABS(aexp)
35	ADR(svar)
37	ASC(sexp)
34	ATN(aexp)
79	*BUMP(pmnum,aexp)
37	CHR$(aexp)
33	CLOG(aexp)
34	COS(aexp)
36-A	*DPEEK(addr)
36-A	*ERR(aexp)
33	EXP(aexp)
40-B	*FIND(sexp,sexp,aexp)
35	FRE(0)
60-A	*HSTICK(aexp)
33	INT(aexp)
38	LEN(sexp)
34	LOG(aexp)
59	PADDLE(aexp)
60-A	*PEN(aexp)
79	*PMADR(pm)
59	PTRIG(aexp)
35	PEEK(addr)
34	RND(0)
34	SGN(aexp)
35	SIN(aexp)
34	SQR(aexp)
59	STICK(aexp)
60	STRIG(aexp)
38	STR$(aexp)
69	*SYS(aexp)
36-B	*TAB(aexp)
36	USR(addr [,aexp...])
38	VAL(sexp)
60-A	*VSTICK(aexp)

EXPLANATION OF TERMS

exp	- EXPression
aexp	- Arithmetic exp
sexp	- string exp
var	- VARiable
avar	- Arithmetic var
svar	- String var
mvar	- Matrix var (or element)
fn	- File Number
line	- line number (can be aexp)
pm	- Player/Missile number (aexp)
[xxx]	xxx is optional
[xxx...]	xxx is optional, and may be repeated
addr	- ADDRess aexp, must be 0 - 65535
<stmts>	one or more statements

NOTE: keywords denoted by an asterisk (*) not in Atari Basic.

APPENDIX L

BASIC A+ MEMORY USAGE

This section describes memory usage INTERNAL to the BASIC A+ interpreter, in what was ROM in the Atari Basic cartridge. See the memory map (appendix D) and memory locations (appendix I) for RAM locations.

Throughout this section, hex addresses are used exclusively. Whenever three addresses are given together separated by slashes (e.g., 4000/6000/8000) they represent the three values associated with systems which have 32K, 40K, and 48K bytes of free RAM available.

CHARACTER GRAPHICS RESERVED AREA 4000/6000/8000

1K bytes of memory are reserved for character graphics. By reserving this memory at fixed locations (at least for any given machine size), the task of writing character set manipulators is greatly reduced.

P.S.: You can find the address of this area via the following subterfuge:

```
  Charactergraphicsaddress = (PMADR(0)-9000)&(14*4096)
```

NOTE: if you do not intend to use character graphics, you can use this area for assembly language routines, etc.

COLDSTART 4400/6400/8400

Where BASIC A+ comes upon loading from disk. Entering at this address performs the equivalent of a NEW.

WARMSTART 4403/6403/8403

Equivalent to where Atari Basic goes when the RESET key is used. Does not destroy any program, but does close files, etc.

JUMP TO TEST FOR BREAK 4406/6406/8406

BASIC A+ checks for the user's use of the BREAK key at the end of executing each line. Exotic driver's might make use of this fact to cause pseudo-interrupts to BASIC A+ at this point. Write for more details, but otherwise don't touch this.

THE SET/SYS() DEFAULT VALUES 4409/6409/8409

Upon execution of NEW, the set of 10 default byte values (SET 0 through SET 9) are moved from this location to 'RAM'. If you would like to change a default, POKE these default values and then save BASIC A+ via CP/A. 4409 (etc.) is SET 0, 440A is SET 1, etc.

CURRENT TOP OF BASIC A+ approx. 7800/9800/B800

But we expect to add features, so if you wish to customize BASIC A+ in this area we suggest you work from the next address(es) down:

DEFINED TOP OF BASIC A+   7B00/9B00/BB00
This is where Players from Player/Missile Graphics start in PMG.1 mode. Also, the area from 7C00/9C00/BC00 up is used by Atari's OS ROM upon RESET and power up to initialize the graphics screen.