JACKSON LANGUAGE SYSTEM BASIC

First Things First

Warranty

The JLS BASIC and accompanying programs are provided "as is" without warranty of any kind, either expressed or implied, including, but not limited to the implied warranties of merchantablility and fitness for a particular purpose. The entire risk as to the quality and performance of the programs is with you. Should the system of programs prove defective, you (and not Jeffrey Glen Jackson) assume the entire cost of all necessary servicing, repair or correction.

The Manual

The manual is written as a reference manual, not as a beginner's guide to programming. It presupposes that the user is familiar with ATARI DOS and ATARI BASIC. This is not unreasonable as all owners should have those items & manuals. Before booting up the first time, I suggest that chapters 1, 2, 3A, 4D, 4E be read to give the user an overview of the design and functioning of the software system, as well as a feel for the layout of the manual itself. Before using BASIC, read 5, 4A, 4B, 4C, and then 3B. Again, this will give you a feel for the language and its features before you actually use it.

The BENCHmark

The file BENCH.OBJ is the compiled screenfill benchmark used by ANALOG magazine. LOAD it and run it (after reading the manual, which explains how). The following chart compares several languages tested by ANALOG and myself:

Language Time (jiffies)

```
ATARI BASIC (token)   4025
MSBASIC (token)       3348
BASIC/A+ (token)      2717
Datasoft/FP (ML)      2435
Draper Pascal (pcode) 2186
Atari Pascal (pcode)   653
Monarch ABC (pcode)    565
*** JLS BASIC (pcode)  279 ***
Datasoft/int (ML)      218
```

```
Action! (ML)            32
BASM (ML)               18
```

Licensing

JLS Basic is now freeware. You may download it from my website for free. I would prefer you didn't put it on your own web site because I would like to see the hit count on my own page.

Correspondence

My web page is http://www.jeff-jackson.com.

My email address is mailto:jeff@jeff-jackson.com.

Trademarks, Copyrights, and References

DOSXL, OS/A+ and MAC/65 are trademarks of OSS, Inc.
DOS 2.0 FMS is (c) copyrighted 1982 by OSS, Inc.
ATARI DOS, ATARI BASIC, and ATARI are trademarks of ATARI, Inc.

All portions of JLS not covered above are (c) copyright 1985, 2000 by Jeffrey Glen Jackson

The following books have been used for references:

ATARI BASIC Reference Manual
OSS BASIC A+ Reference Manual, (c) 1982, 1981, OSS, Inc.
ATARI Personal COmputer System Operating System User's Manual, (c) 1982, ATARI, Inc., (C016555).
ATARI Personal Computer System Hardware Manual, (c) 1980, ATARI, Inc. (C016555)
DISK OPERATING SYSTEM II Reference Manual, (c) 1981, ATARI, Inc. (C016347 REV. 1)
ATARI 850 Interface Module Operator's Manual, (c) 1980, ATARI, Inc. (C015953 Rev. 1).
Inside ATARI DOS, Bill Wilkenson, (c) 1982, Small System Services, Inc.
A Reference Manual For OS/A+, (c) 1982, OSS, Inc.

I. Overview

A. Design

JLS is an integrated program development system. Its major innovation for the ATARI world is the concept of modules and the manner they are implemented. A module is simply a compiled program than can use and/or provide external references. The CP (Control Program, Consol Processor, Monitor, Disk Utility Package, or whatever else it might be called) of JLS can maintain more than one module in memory at a time. When CP loads a module from disk, it relocates it at LOMEM (the lowest unused address) and moves the LOMEM pointer up. Modules are deleted (killed) from memory in LIFO (Last-In-First-Out) order. That is, they form a stack.

There are two types of modules. The first is the utility. Once it is loaded, it can be run just like it was a built-in command. A utility might use subprograms (a subroutine with parameter passing) that are not actually contained in the module. Each of these subprograms has a name. When CP loads a module, it resolves these external references (if possible). The subprograms can come from either of two sources. The CP itself has some, or they can be in another type of module called a package. A package is a module whose primary function is to provide subprograms for other modules to use. Hence several utilities that are in RAM at once can share only one copy of the subprograms. The package must be loaded before it can be used to resolve references. More importantly, modules written in say BASIC will be able to be linked in this manner to modules written in another language such as PASCAL (which is the next language to be supported).

The compilers output a stack oriented pcode which is interpreted by the program RTL.SYS (Run Time Library) which is loaded in at boot time. In fact, the CP is written in BASIC and compiled. All files with the extender .SYS must be present on the boot disk for the system to bootup. DOS.SYS is the first file to be loaded. It contains the DOS 2.0 FMS (File Management Subsystem), the memory manager, and a loader to install the other system files. CONFIG.SYS contains a list of those files to be loaded in. Minimally this includes CP.SYS, RTL.SYS, and RELOCATE.SYS. These are installed in the RAM shadowing the ROM on the 64K XL computers.

B. Commands

Once the system is booted up, the copyright notice is displayed and you are prompted with a $ (actually, before you are prompted, the batch file STARTUP.BAT is ran -- but this is discussed later). The $ is your prompt to enter commands for the CP to execute. There are two classes of commands: intrinsic (those built into CP) and extrinsic (those executed by loaded modules).

Intrinsic commands' syntax is generally more restricted than extrinsic ones'. This is because CP is limited to 4K while extrinsic commands have all of memory to use. Chapter two describes each intrinsic command. The syntax specification of each command shows the $ prompt, the command name and its argument. Arguments are usually file or module names and must be seperated from the command by one or more spaces. Not all commands have an argument, and the argument of some commands is optional. If an argument is optional, its one-word description in the syntax specification is enclosed in [brackets].

Usually extrinsic commands are far more flexible (syntax-wise) than intrinsic ones. Most of them that make extensive use of the command line use the package COMMAND. COMMAND supports up to 4 positional arguments and 8 options. Positional arguments are things like filenames of a copy command (COPY D2:FROM.BAS TO.*). Required positional arguments that are omitted are prompted for. Obviously, if there are say 3 arguments, you cannot omit only the second one. You may omit the 3rd, or the 2nd and the 3rd, or all three.

Often, when you are prompted, a default value is also shown with the cursor positioned on its first character. Just hit RETURN to use it. Also, be sure to write over the entire default should you supply a different value. For example, if the default is 100 and you are supplying 10, be sure to press the space bar to get rid of the second zero after you type in 10.

Options on an extrinsic command can come anywhere after the command and in any order. They can be placed before, in between, or after the arguments. Note that this is far more flexible than the intrinsic options. (In version 2, there are no options on intrinsic commands)

There are three formats of options:

```
    /keyword=value
    /keyword
    /NO_keyword
```

Actually, the last two are equivalent to:

```
    /keyword=1
    /keyword=0
```

respectively. Generally, keywords ar limited to 8 characters and value is limited to 20 characters. There may be no spaces within the option, but it must be seperated from other options, arguments and the command by one or more spaces.

Several commands may be issued on one line by seperating them with a space, semicolon, space. If an extrinsic command is not already LOADed, it will be LOADed, executed, then KILLed. If the module name doesn't include a file spec the DEF location is searched first, then the OBJ location (see the DEF and OBJ commands).

Chapter III describes the provided modules. The module documentation is segregated (may the ACLU forgive me) between utilities and packages. The syntax of extrinsic commands is specified in the following manner. The first line has a $, the command name, and a short, one word description of each positional argument (in proper order of course). Each of the following lines show the default setting of each option. (If the default setting is what you want, you can omit that option when you type in the command). Below that is a list of any packages that the utility requires to be loaded in before it is to resolve its references. The package documentation just shows a simple load command. Remember that the load command might need a device specification and/or an extender. See LOAD in chapter 2. Below that are the BASIC declarations of all the subprograms. Finally, each subprogram is described.

C. The DOS (FMS)

This system currently uses DOS 2.0. This is the exact same FMS used in ATARI DOS II and OS/A+ V2.0, and can read and write files made by OS/A+ V2.1, DOS/XL V2.2, DOS/XL V2.3, and probably several other DOS's as well. (The FMS is copyrighted by Optimized Systems Software. OS/A+ and DOS/XL are tradmarks of the same.) Disk filespecs consist of the letter D, followed by a digit (optional), followed by a colon, followed by the filename (up to 8 letters/digits), and followed by an optional extender (a period followed by up to 3 letters/digits). Most commands will use the default device if it is omitted. Some will supply a default extender. To specify no extender when there is a default extender, give the extender as a period only (eg. FILE.). Note that this DOS is not compatibile with "medium" density disks created with DOS 2.5.

Wildcards can also be used. There are two wildcards **?** and *****. The **?** substitutes a single letter; ***** substitutes the rest of either the filename or the extender. The wildcards have two uses. The first use is to specify multiple files. The prime example of this is in the **DIR** command. For example, **DIR *.BAS** will list all files with **BAS** as the extender. **DIR COMMAND.*** will list all files with the filename **COMMAND** and any extender. **DIR**

B?D*.??A will list all files such that their filename's first letter is **B** and third letter is **D** and the third letter of the extender is **A**, regardless of the other letters.

The other use is epitomized by the **COPY** command. The **COPY** command has two arguments, the first is the source file and the second is the name of the file the source is to be copied to. For example, **COPY TEST.BAS TEST.BAK** will create a copy of the file **TEST.BAS** that will be called **TEST.BAK**. If you use wildcards in the second (destination) file, those wildcards are substituted with the corresponding characters from the first (source) file name. Thus **COPY TEST.BAS *.??K** will do the exact same thing. COPY also permits both types of meanings in wildcards. **COPY *.BAS *.??**K will make a copy of all files that have the extender BAS such that the copies will have the same filename but the extender BAK.

II. The Control Program

A. Intrinsic Commands

$ CLS

The **CLS** command performs a **GRAPHICS 0**. This is useful for restoring the screen after a stack overflow.

$ DEF [*device*:]

The **DEF** command is used to examine or change the current default device for filenames. If **DEF** is typed in by itself, the default device is shown. The default device is **D**: by default. If you put a ***device*:** specifier after the command the default device is changed to that device name. Note that you must include the colon after the name, but the device number is not necessary (It defaults to 1). See also the **OBJ** command.

$ DIR [*filespec*]

The **DIR** command allows you to examine the **DIR**ectory of files on a disk. If filespec is omitted, it defaults to ***default*:*.*** where ***default*** refers to the default device (see **DEF** this section). If you specify just the device (with the colon) the filespec used is ***device*:*.***. If you omit a device, but supply a ***filename*[*.ext*]**, that is used with the default device to construct the filespec. No default extender is supplied if you omit just the extender. The **DIR** command uses the filespec to selectively list files on the disk. Only files with the specified characters in their particular position are listed. The characters **?** and ***** are wildcard characters (see "Wildcards" in Chapter 1). To obtain a printed listing of files, see the **CATLG** and **FILES** extrinsic commands in Chapter 3.

$ ERASE *filespec*

The **ERASE** command erases files matching the given filespec (i.e., the same files listed with **DIR** filespec). No verification for individual files is requested. For this feature, see the **DELETE** extrinsic command in Chapter 3.

$ FLOAD *filespec*

The **FLOAD** command forces a **LOAD** of a module even if there is already a module of the same name in the module directory. See also **LOAD**.

$ KILL *module*

The **KILL** command removes the specified module from memory. It also removes all modules that were loaded in after the specified module since the modules are organized as a stack. If there is more than one module of the same name loaded, only the most recently loaded one will be killed.

$ LOAD *filespec*

The **LOAD** command loads in a module from disk, relocates it at the bottom of memory, resolves its external references, and adds its name and its external entry points to the module directory. If the device is omitted, the default device is used (see **DEF** this section). It it isn't present on the **DEF**ault device, then **OBJ**ect device is tried.

The name that the module is given is the same as the filename portion of **D*n*:*filename.ext***. The extender defaults to **.OBJ**. Beware of using wildcards: only one module will be loaded and it may recieve an unexpected name. The **LOAD** command will not load a module that is already loaded. That is, it will not create a duplicate name in the module directory. If you must do that, use the **FLOAD** command, which see.

If the specified modules has unresolved references, the module will not be **LOAD**ed. Extrinsic commands may also be ran without **LOAD**ing them. They are **LOAD**ed, ran, then **KILL**ed automatically when used in that manner.

$ MDIR

The **MDIR** command displays a list of the modules that have been loaded into memory with the most recently loaded ones at the top. It also displays the load address, then the size of each module after its name. At the end of the listing is the number of free bytes of memory left.

$OBJ [D[*n*]:]

The **OBJ** command specifies a default place to find object files when **LOAD** fails to find them in the **DEF**ault location.

$ PROTECT *filespec*

The **PROTECT** command protects (or locks) the specified file(s with wildcards) from being deleted or altered.

$ RENAME *filespec*,*filename.ext*

The **RENAME** command renames the files specified in ***filespec*** with the name specified in ***filename.ext***. Wildcards in ***filename.ext*** are replaced with with the corresponding characters in the ***filespec***.

$ REPEAT

The **REPEAT** command is intended for use with the semicolon in defining multiple statements on a command line. It causes the command line to be reexecuted again. For example:

$ PAUSE ; BASIC *x* ; PAUSE ; EDIT ; REPEAT

will let you alternate between editing and compiling.

$ UNPROTECT *filespec*

The **UNPROTECT** command undoes the protection provided by the **PROTECT** command (which see). It is OK to unprotect a unprotected file just like it is OK to protect a protected file.

$ XDIR

The **XDIR** command displays a list of the modules that have been loaded into memory with the most recently loaded ones at the top. It also displays the load address, then the size of each module after its name. At the end of the listing is the number of free bytes of memory left. It also displays the external entry points and their addresses. They are indented underneath the module which they are a part of. Note that after the last module (and maybe its external entry points) there is a blank line and then some more external entry points. These entry points are into the Control Program itself and are always there.

B. Batching

A batch file is a file that contains a list of commands to be executed. A batch file may contain any of the above intrinsic commands, any loaded extrinsic commands, and any of the below commands that are geared for batching. When the system is first booted up, the batch file **STARTUP.BAT** is executed, if it is present on **D1:**. Since some modules will require several other modules to be already loaded in before it is loaded to resolve its external references, it would be convenient to use a batch file to load all of them in. Since some of the required modules might already be loaded in, such a batch procedure should use the **LOAD** command rather than **FLOAD** to prevent multiple copies of the same module from being loaded. All batch files should end with the **END** command (which see), though everything will work OK if it is omitted. The lines of a batch file should begin with a **$**.

$ BATCH *filespec*

The BATCH command initiates the exection of the specified batch file. The device of the filespec will default to the default device, and the extender will default to .BAT.

$ END

The **END** command deactivates batch mode so that the next command will be requested from the keyboard.

$ NOSCREEN

The **NOSCREEN** command deactivates the batch command echo to screen feature. By default, each batch command is not printed on the screen as it is read in. See **SCREEN** to activate this feature.

$ NOTRAP

The **NOTRAP** command causes the control program to remain in batch mode after errors are raised. This is the default state. Since **CAR** can result in an extraneous error, you should be in this mode when **CAR** is executed from a batch file.

$ PAUSE [*text*]

The **PAUSE** command rings the buzzer, displays the text, and waits till the user presses RETURN before continuing. The text begins with the character after the space after the **PAUSE** command. The text may contain any character except semicolon as it is not subject to the usual command line syntax.

$ PRINT [*text*]

The **PRINT** command acts like the **PAUSE** command in that it displays the text, but it doesn't ring the buzzer or wait for the user to press RETURN.

$ REM *text*

The **REM** command lets you stick remarks in your batch file. It doesn't actually do anything.

$ SCREEN

The **SCREEN** command causes each subsequent batch command to be displayed on the screen as it is read in. This is not the default case. See **NOSCREEN** to deactivate this feature.

$ TRAP

The **TRAP** command causes batch mode to be deativated should any command result in an error condition. The default is that batch mode continues after errors. Beware of **CAR** in **TRAP** mode. See also **NOTRAP**.

III. Provided Modules

A. Utilities

$ APROPOS *word*

Searches **HELP.HLP** for ***word*** and displays topic name and line containing ***word***. See also **HELP**.

$ BOOT

The **BOOT** command reboots the system. This is preferable to turning the system off and on. Remember to hold down on the OPTION consol button all the time the screen is black so you won't get stuck with ATARI BASIC eating 8K of your RAM.

$ C0
$ C2

The **C0** and **C2** commands set the left margin to column 0 and 2 respectively. The OS by default sets it to column 2 for TVs that overscan. People whose TVs don't overscan will probably prefer column 0.

$ CAR

The **CAR** command runs the cartridge, if it is installed. This command is of limited usefulness as this system is more standalone and not designed for use with ATARI BASIC or any currently existing cartridge (except perhaps MAC/65). Exiting from a cartridge will sometimes cause a random error message (MAC/65 is bad about this). This message should be ignored (See **NOTRAP** in "Batching" section of this chapter). When DOS is returned to from a cartridge, then the cartridge is reentered, your cartridge-program will probably be erased.

$CAT *filespec*
/NO_QUERY

Copy text files, one line at a time, to the screen. The **/QUERY** option will ask for verification for each file matching ***filespec***.

$ CATLG ***filespec1 filespec2***

The **CATLG** command displays a catalog of your files on disk. It acts just like the intrinsic command **DIR** ***filespec1***, except that the output is routed to the ***filespec2*** instead of the screen.

$ CONFIG [*unit density*]

(requires **IO**)

Issuing **CONFIG** with out any arguments will display the density (or error status returned) of each disk that DOS is configured for. It may also be used to change the density of PERCOM compatible disk drives if you give the two positional arguments:

unit = a digit **1** to **8** specifying which disk unit.

density = which density: **S** for single **D** for double.

$ COPY ***filespec1 filespec2***
/NO_WAIT
/NO_SINGLE
/NO_QUERY
/NO_APPEND

(requires **COMMAND**)

The **COPY** command copies the files specified by ***filespec1*** to the files specified by ***filespec2***. If multiple files are specified in ***filespec1*** by using wildcards, it will act as though seperate **COPY** commands were issued for each source file. Wildcards in ***filespec2*** are replaced with the corresponding characters from ***filespec1*** (just like **RENAME**). If the destination file already exists, you will be asked whether it is OK to be replaced. Type **Y** for yes or **N** for no. There are four options that may be specified:

/WAIT causes **COPY** to prompt you to insert the source disk before the command begins. Press RETURN when it is mounted.

/SINGLE informs **COPY** that you are making the copy using a single drive. You will be prompted to insert the destination and source disks as each is needed. Press RETURN when the proper disk is mounted.

/QUERY causes **COPY** to request verification before copying each file. Respond **Y** for yes or **N** for no. A "no" response will cause **COPY** to just go on to the next file.

/APPEND causes **COPY** to append each source file to its destination file. The destination file must already exist.

Each of the above options may be abbreviated to a single letter (**/W /S /Q /A**). See also **SDCOPY**.

$ CR2EOL ***filespec***

Transform ASCII CR to ATASCII EOL in the specified file.

$ DISKCOPY ***source destination***

(requires **IO**)

The **DISKCOPY** command copies an entire disk. The ***source*** and ***destination*** are specified using a single digit (**1-8**) which is the drive number. You are prompted to insert the source disk regardless of anything else. Press RETURN when it is mounted. If the ***source*** and ***destination*** are the same drive, you are prompted to insert the proper disk into the drive as needed.

$ EDIT [*file*]
/LINES=512

The **EDIT** command runs a program text editor that can be used to edit BASIC programs, batch files, or any text files. It is similar to ATARI BASIC and the Assembler/Editor's editors. You are prompted with **EDIT** and may enter **EDIT**'s commands on any line.

If ***file*** is specified, it is automatically **LOAD**ed. Note that **LOAD**ing will check the first line to see if it has a line number. If not, an implicit **GET** is done instead.

In the below descriptions, the format **[*from* [*to*]]** has the following defaults:

omitted = first line to last line
from = ***from*** only

Also note that either a number of spaces or commas can be used to seperate the arguments of the subcommands.

The **/LINES** option will change the maximum number of lines from the default of 512 to the specified count.

line#

deletes that line.

line# text

inserts/replaces a line.

DELETE *from* [*to*]

deletes a range of lines (***to*** defaults to ***from***).

DOS

returns to the control program. Will warn if program modified and give opportunity to **SAVE** or **PUT** if file was **LOAD**ed or **GET**ed.

FIND *text*

displays all lines that contain the specified text. The text begins with the character after the space after **FIND**.

FREE

displays number of free bytes and of lines used, and the maximum number of lines permitted. See also **MERGE** and **SAVE**.

GET *filespec*

loads in the specified file assuming it does not have line numbers. The lines are numbered starting at 1000 and incrementing by 10. See also **PUT**.

LIST [*from* [*to*]]

lists the specified ranges of lines onto the screen.

LOAD *filespec*

loads the specified file into memory. It replaces any program already in memory. The lines are checked to see if they are numbered. If not, a **GET** is done instead..

MERGE *filespec*

loads the specified filespec into memory without erasing the one already there. The lines of the file must be numbered. See also **LOAD** and **SAVE**.

NEW [*max_lines*]

erases the program in memory. By default **512** lines are the maximum the editor will handle. New can also be used to change this number if you specify a number after the command. Typing **NEW** without a number after it will leave the maximum number of lines alone. When you exit to the CP and return to **EDIT**, the maximum number of lines will be remembered. **KILL**ing and re**LOAD**ing **EDIT** will cause it to revert to the default.

PUT ***filespec* [*from* [*to*]]**

saves the file in memory, but does not output the line numbers with the lines. See also **GET**.

REN [*start* [*increment*]]

renumbers the lines of the program in memory starting with the number start and incrementing by increment. ***Increment*** defaults to ***start*** and ***start*** defaults to **10**. Generally you should not renumber BASIC programs because **REN** does not check **TRAP**s, **GOTO**s, **THEN**s or any other line number references.

SAVE ***filespec* [*from* [*to*]]**

saves the program in memory to a file. It saves only the specified line range and includes the line numbers with the lines. See also **LOAD**, **MERGE**. Like most other provided commands you may omit the device from filespecs and let it default to the default device.

$ *command*

You may execute intrinsic CP commands by typing in a '**$** ' followed by CP command line. This is useful for getting a directory, protecting and unprotecting files, erasing files, ending batch mode (with an **END**), etc. Note, you should not **LOAD** or **KILL** modules or run modules.

$ EOL2CR *filespec*

Transform ATASCII EOL characters to ASCII CR characters in the specified file.

$FILES [filespec]

$FILES [*filespec1*]
/OUTPUT=E:
/LABEL=
/DATE=

/INIT=
/NUMBER=

FILES is much like **CATLG** except that it sorts the list of files and if the output device is **E:**, it will pause and wait for you to press a key before proceeding every 20 lines of output.

$ FORMAT *drive*

The **FORMAT** command formats a disk. You are prompted to mount the disk to be formatted. Press RETURN when you have done so. The ***drive*** is specified using a single digit (**1-8**).

$ HELP *topic*

Display a help screen for a topic. If multiple help screens exist for a given topic, the user is prompted to press a key between screens.

HELP works by finding a **HELP.IDX** file at **LOAD** time, searching first the DEFault path, and then the OBJect path. It loads this index into memory just the one time. When the command itself is executed, it reads the help screens from a file called **HELP.HLP** found on the same device as **HELP.IDX**.

If **HELP.HLP** is copied to another location, it is necessary to rebuild **HELP.IDX** with the **HELPIDX** command. See also the **APROPOS** command.

$ HELPIDX [*filespec*]

Rebuild **HELP.IDX** file from **HELP.HLP** file.   Alternative help file may be specified. This is necessary if ***HELP.HLP*** is copied or moved since the index contains absolute sector addresses of help screens.   See also **HELP**.

$ MENU

????????????????????????

Experimental menu driven interface that provides special versions of the routines in **COMMAND**.   Not quite ready for prime time.

$ PUTDOS

The **PUTDOS** command writes a **DOS.SYS** file to a disk. You are prompted for the drive allocation bits, the number of buffers the DOS is to support, and the disk unit number (**1-8**) DOS is to be written to. The values for drive allocation bits are:

1	1 drive	31	5 drives
3	2 drives	63	6 drives
7	3 drives	127	7 drives
15	4 drives	255	8 drives

Each single density drive requires one buffer; each double density drive requires one buffer. Each open single density file requires one buffer; each open double density file requires two buffers. The system, as shipped supports two drives and eight buffers. On a double density system, this is enough for two disk files to be open at the same time.

$ SDCOPY *filespec1 filespec2*
/NO_DS
/NO_WAIT
/NO_QUERY
/NO_APPEND

(requires **COMMAND**, **IO**)

The **SDCOPY** command is used to transfer files from a single density disk to a double density disk (or vice versa) using only a single PERCOM compatible disk drive. The device of both filespecs must be **D1:**. If you have two drives, use **COPY**. See **COPY** for use of wildcards, **/WAIT**, **/QUERY**, and **/APPEND**.

/DS specifies that the transfer is to be from double to single density rather than the default which is the other way around. **/SINGLE** may not be specified, but is always implied since **SDCOPY** uses disk drive one. See **COPY**.

$ SET
/LEFT=2
/RIGHT=39
/CHAR=202
/BACKGND=148
/BORDER=0
/FINE=0
/DELAY=48
/REPEAT=6
/CLICK=0

(requires **COMMAND**)

SET allows the user to change certain system variables. Up to eight may be changed in one command. The default values shown above are not the actual defaults. All defaults are the current values. The values show above are the values after a reset.

/LEFT=*n* /RIGHT=*n* set the left and right margins.

/CHAR=*n* /BACKGND=*n* /BORDER=*n* set the character, background, and border colors. The next **GRAPHICS 0** undoes these settings.

/FINE=1 will enable fine scrolling the next **GRAPHICS 0** (use **CLS**). Fine scrolling may not work well because of a bug in ATARI's OS.

/DELAY=*n* sets the delay (in 1/60ths seconds) before autorepeat on the keyboard begins.

/REPEAT=*n* sets the delay between repeats for keyboard autorepeat.

/CLICK=1 inhibits the keyboard click.

See also **SHOW**.

$ SETTIME [*hh*:*mm*:*ss*]

The **SETTIME** command sets the system clock. Hours, minutes, and seconds must all three be specified, separated by colons, and exactly two digits each. See also **TIME**.

$SETUP

Scans **D1:** for batch files with single letter extensions and presents them in a menu. Usefull as the last command of a **STARTUP.BAT** to select one of multiple startup scripts.

$ SHOW [LEFT] [RIGHT] [CHAR] [BACKGND] [BORDER] [FINE] [REPEAT] [DELAY] [CLICK] [DEV] [VERSION] [DOS]

The **SHOW** command shows the values of several system variables. The variables may be specified in any order after **SHOW** and their is no limit to the number that may be specified (as long as everything stays on one logical line).

LEFT, **RIGHT**, **CHAR**, **BACKGND**, **BORDER**, **FINE**, **REPEAT**, **DELAY**, and **CLICK** are the same variables that may be set with the **SET** command.

DEV is a list of device drivers installed.

VERSION is the current JLS version number.

DOS shows the current drive allocation bits and buffer allocation.

$ SURROUND

A simple one or two player game. Requires joysticks.

$ TIME

The **TIME** command shows the current time. See also **SETTIME**.

$ TYPE ***filespec***
/HEADING=
/NO_QUERY
/OUTPUT=E:
/LINESIZE=80
/FORMSIZE=66
/TOP=6
/LEFT=5
/INDENT=2
/INIT=

(requires **COMMAND**)

The **TYPE** command makes nice printouts of source code. Its options let you set the line and form size and the top (and bottom) and left (and right) margins as well as output escape codes to configure the printer. filespec specifies which files are to be printed out. Wildcards may be used to specify several files. Each file will start on a seperate page.

/NO_HEADING causes the heading to be suppressed. Otherwise, the ***filespec*** of the file being typed and the current page number are put on the first line which is then followed by a blank line.

/QUERY causes **TYPE** to prompt you whether you want to type each file or not. Respond **Y** to type it and **N** to go on to the next file.

/OUTPUT=*filespec* specifies the output file. The filespec may not contain wildcards. It is opened only once. The initialization characters are sent only once (see **/INIT=**). If you specify a disk file, then want to print that out later, use **COPY** to make that printout (weird things might happen if you use **TYPE**).

/LINESIZE=# specifies the number of characters there are per line. **80** is the usual for pica, **96** for elite.

/FORMSIZE=# specifies the number of lines per page (or per form). **66** is the usual (6 lines/inch).

/LEFT=# specifies the number of characters in the left and right margin. (These are subtracted from the linesize above - resulting in 70 characters per line actually printed in the default).

/TOP=# specifies the number lines in the top and bottom margins. **6** usually creates a one inch margin.

/INDENT=# specifies the number of characters to indent when a logical line exceeds one physical line. Indention occurs on the second and subsequent physical lines.

/INIT=*characters* specifies characters to be output to the printer to configure it when it is opened for output. This can be used to change it to elite, bold characters, 88 lines/page, etc. The characters may not include spaces.

$ VERIFY [ON|OFF]

The **VERIFY ON** command causes all disk writes to be verified. This is slower than **VERIFY OFF**, but is much safer. This is the default. **VERIFY** by itself reports the current VERIFY status.

B. Packages

$ LOAD COMMAND

BASIC declarations:

```
RECPROC P$DIR0(STRING(24))
RECFNC(STRING) F$DIR1,F$MATCH(STRING(*),STRING(24))
RECPROC P$SCAN0
RECFNC(STRING) F$SCAN1(STRIG(*),STRING(*))
RECFNC(STRING) F$POS(BYTE,STRING(*),STRIG(*))
RECPROC P$SCAN2
```

P$DIR0(*filespec*)
F$DIR1-->*filespec*

The procedure **P$DIR0** sets up the package for reading the directory from channel 5. You should not use **OPEN** under certain circumstances to do this because if you open another channel to disk while a directory channel is open, the directory channel becomes confused. The use of **P$DIR0** and **F$DIR1** bypasses this problem. In addition, the string returned by **F$DIR1** is converted to the usual Dn:filename.ext form where Dn: is the default device. **F$DIR1** returns null string when there are no more filenames.

F$MATCH(*filespec1*,*filespec2*)-->*filespec3*

The **F$MATCH** function is used to implement the second type of wildcards (see Chapter I section C). ***Filespec1*** should not have any wildcards in it. The value returned (***filespec3***) is the same as ***filespec2*** except that the wildcards in ***filespec2*** are substituted with the corresponding characters from ***filespec1***. ***Filespec3*** will have the same device as ***filespec2***, or the default if ***filespec2*** didn't have one.

P$SCAN0

The procedure **P$SCAN0** must be called before the following functions and procedures. It collects information on up to four positional parameters and up to twelve options. The following functions and procedures use that information.

F$SCAN1(*keyword*,*default*)-->*value*

The function **F$SCAN1** returns the value of the specified option. **Keyword** signifies which option (it does not include the **/** (slash) or the **=** (equal sign) or the **NO_** portions of an option. If the option was omitted from the command line, the **default** value specified as the second parameter is returned. Otherwise the characters after the equal sign in option are returned. There are two special cases. **/*keyword*** by itself is seen as the same as **/*keyword*=1**, and **/NO_*keyword*** is seen as **/*keyword*=0**. The default value should be set to "**0**" or "**1**" accordingly.

F$POS(*position*,*prompt*,*default*)-->*value*

The function **F$POS** fetches the value of positional arguments. The first argument of this function specifies which positional argument is being fetched. They are numbered **0**,**1**,**2**, and **3**. If that positional argument is omitted, the ***prompt*** and ***default*** is displayed, and the cursor is placed over the first character of the ***default*** so that the user need only press RETURN to accept the default value. Be careful that the prompt and default can both be displayed on only one line. If the ***prompt*** is an empty string, then the ***default*** is returned without prompting the user.

P$SCAN2

The procedure **P$SCAN2** should be called after the values of all positional arguments and options have been determined. It generates error **63** and messages for any argument or option on the command line that was not accessed; that is, that was extraneous.

CP's external entry points

BASIC declarations:

```
RECFNC(STRING) F$DEF(STRING(*))
RECFNC(STRING) F$WORD
PROC P$CMD(STRING(*))
RECFNC(BINARY) F$ALLOC(BINARY)
```

These functions are part of the code of CP, and here therefore always available. They are the entry points listed after the blank line by the XDIR command.

F$DEF(*string1*)-->*string2*

The **F$DEF** function adds the default device to the file specification ***string1*** (if ***string1*** does not already have one). Nothing is done to the string of there is already a device specified.

F$WORD-->*string*

The **F$WORD** function returns the next word from the command line. A word is any combination of characters delimited by spaces, or a space and a carriage return. The word is limited to 32 characters. Normally, when you use the command line, you will use the **COMMAND** package. **COMMAND** uses this function to access the command line.

P$CMD(*string*)

The **P$CMD** procedure executes a CP command. The string may optionally begin with a **$**. Extreme caution should be used in calling this to maintain system integrety. Whether it returns to your program or not depends on what you ask it t do:

PROTECT, **UNPROTECT**, **ERASE**, **RENAME**, **DEF**, **END**, **SCREEN**, **NOSCREEN**, **MDIR**, **BATCH**, **TRAP**, **NOTRAP**, **PAUSE**, **PRINT**, **CLS**, **REM**, **DIR** will let **P$CMD** return to your program.

Extrinsic commands will never return to your program.

LOAD, **KILL** may not be done. System integrity will be violated if they are attempted.

DIR results in **TRAP 0** being active.

BATCH will not actually execute a batch file. It will enable batch mode. When your program ends, that batch file will be activated. This is the only way for a program to **LOAD** and **RUN** a program. It must create a batch file, call **P$CMD("$ BATCH *filename*")**, then **END**.

END will deactivate batch mode. If your program was run with a batch file, this command will deactivate that batch mode.

All calls to **P$CMD** that are not expected to return should have a **STOP** statement immediately after it. This is because Version 3.0 of the CP might let some of these return. If you don't, you will run into problems if you forget when you recompile under Version 3.0 of the JLS Language System.

Calling **P$CMD** to do an extrinsic command is a convenient way to tranfer control to another module. That module must already be **LOAD**ed in. See **BATCH** notes above for how to transfer control to a module not already **LOAD**ed in. (This might not be true anymore now that trying to run a module that isn't **LOAD**ed will result in the module being **LOAD**ed, ran, then **KILL**ed).

F$ALLOC(*binary1*)-->*binary2*

The **F$ALLOC** function should be called from within the **ATTACH** subprogram only. Doing otherwise would violate system integrety. This function allocates ***binary1*** more bytes to the module calling it (which module is now being **LOAD**ed in since **ATTACH** only should call it) and returns the address of that block of memory. An example of a module that uses it is **RS232**. **RS232**'s **ATTACH** downloads the device driver to the address that will be returned as **binary2**, computes how big the driver is, and requests that storage be allocated to the module.

$ LOAD GAME

BASIC declarations:

```
FNC(BYTE) PADDLE(BYTE),STICK(BYTE)
FNC(BYTE) PTRIG(BYTE),STRIG(BYTE)
FNC(BYTE) VPEN,HPEN,VSTICK(BYTE),HSTICK(BYTE)
FNC(BYTE) OPTION,SELECT,START,HELP
FNC(FLOAT) RND,RAND(FLOAT,FLOAT)
FNC(BINARY) RND2,RAND2(BINARY,BINARY)
FNC(BYTE) RND1,RAND1(BYTE,BYTE)
PROC SETPOLY17(BYTE)
PROC SETFIN15(BYTE),SETFIN(BYTE,BYTE)
PROC SETFILTER(BYTE,BYTE)
PROC SOUND(BYTE,BYTE,BYTE,BYTE)
PROC SOUND8(BYTE,BYTE,BYTE,BYTE)
PROC SOUND16(BYTE,BINARY,BYTE,BYTE),SOUNDOFF
```

The **GAME** package consists of three classes of subprograms: (1) Game controller support, (2) Random numbers, and (3) Sound support. The Random number class is exact same set of routines provided in the **MATH** package. The user should refer to that package for their documentation. The user may need to refer to the hardware manual for some of the technical sound generation documentation, or experiment a lot.

PADDLE(*number*)-->*position*

The argument and value returned by this function is the equivalent to the **PADDLE** function of ATARI BASIC. The paddles are numbered **0-7** (or **0-3** and the XLs). The position ranges from **1** to **228** as you turn the knob counterclockwise.

STICK(*number*)-->*position*

Again, this **STICK** function is the same as ATARI BASIC's. The sticks are numbered **0-3** (**0-1** on the XLs). The value indicates direction as per below:

```
  10 14 6
   \ ! /
    \!/
11--15-- 7
    /!\
   / ! \
  9 13  5
```

PTRIG(*number*)-->*trigger*
STRIG(*number*)-->*trigger*

The functions **PTRIG** and **STRIG** return TRUE (**&255**) if the paddle and joystick triggers are pressed respectively and FALSE (**&0**) if not.

VPEN
HPEN

The **VPEN** and **HPEN** return the vertical and horizontal lightpen positions respectively,

VSTICK(*number*)-->*direction*

The **VSTICK** function returns the vertical direction of the joystick. +1 means the joy stick is pushed up. **0** means it is vertically centered. **65535** means it is pushed down. Note that the number **65535** acts like **-1** when used in addition or subtraction. If you need the equivelent in **BYTE** type, **AND** the **VSTICK** value by **255** before applying the **BYT** function to avoid a conversion error. **&255** in **BYTE** will act like **-&1** in addition and subtraction.

HSTICK(*number*)-->*direction*

The **HSTICK** complements **VSTICK**. It returns **+1** when the joystick is pushed right, **0** when centred, **65535** (-1) when pushed left.

OPTION-->*flag*
SELECT-->*flag*
START-->*flag*

These functions return true (**&255**) when the OPTION, SELECT, and START buttons are pushed respectively and false (**&0**) otherwise.

HELP-->*helpkey*

The **HELP** function returns **17** if HELP has been pressed, **81** when SHIFT HELP has been pressed, and **145** if CTRL has been pressed since the last time the **HELP** function was issued. That is, the computer remembers if the HELP key has been pressed from one issuence of the **HELP** function to the next.

SETPOLY17(*value*)

0 specifies 17-bit polynomial in sound generation. **1** specifies 9-bit polynomial.

SETFIN15(*value*)

0 species 63.9210 Khz for the input frequency (Fin). **1** specifies 15.6999 Khz. 15.6999 will result in any given sound being 2 octaves lower in frequency.

SETFIN(*voice*,*value*)

Voice may be **1** or **3**. ***Value*** **0** indicates 63.9210/15.6999 Khz. **1** indicates 1.78979 Mhz. The higher input frequency is useful only for 16 bit frequency resolution.

SETFILTER(*voice*,*value*)

Voice may be **1** or **2**. A high pass filter can be inserted on either clocked by channels 3 and 4 respectively. A *value* of ***0*** specifies no filter, ***1*** specifies a filter.

SOUND(*voice*,*pitch*,*distortion*,*volume*)

The **SOUND** procedure acts just like ATARI BASIC's. It clears all settings made with the previous or subsequent procedures. The valid ranges are: voice **1-4**, pitch **0-255**, distortion **0-14** (even), volume **0-15**.

SOUND8(*voice*,*pitch*,*distortion*,*volume*)

SOUND8 is similar to **SOUND**. The difference is that it does not reset any other settings. However, if voice **1** and **2**, or **3** and **4** are combined together for 16 bit resolution and you specify, for example, voice 1. Voice 1 and 2 would be split, and voice 2 turned off. That is, it will revert the specified voice back to 8 bit resolution.

SOUND16(voice,pitch,distortion,volume)

SOUND16 is similar to **SOUND8** except it combines either voices 1 and 2, or 3 and 4 for 16 bit resolution on the frequency. You may specify only voice **1** or **3**. Voices 2 and 4 will cause an error 63. Generally, you should use Fin of 1.78979 Mhz on 16 bit resolution, else sounds will be too low pitched.

SOUNDOFF

SOUNDOFF turns off all sounds and returns all parameters to their default (0) value.

The below gives the formulas from the *Hardware Manua*l for computing the resulting frequency of any given setting.

Fout=output frequency
Fin=input frequency (either 1.78979 Mhz, 63.9210 Khz, or 15.6999 Khz)
Pitch=the pitch specified in a SOUNDxx procedure.

For 64 and 16 Khz:

```
         Fin
Fout=-----------
     2*(pitch+1)
```

For 1.79 Mhz and 8 bits resolution:

```
         Fin
Fout=-----------
     2*(pitch+4)
```

For 1.79 Mhz and 16 bits resolution:

```
        Fin
Fout=-----------
     2*(pitch+7)
```

$ LOAD HEAP

????????????????????????????

$ LOAD IO

BASIC declarations:

```
FNC(BYTE) DOS$STATUS(BYTE,BYTE(3))
FNC(BYTE) DOS$READ(BYTE,BINARY,BINARY,BYTE)
FNC(BYTE) DOS$WRITE(BYTE,BINARY,BINARY,BYTE)
FNC(BYTE) DOS$CONFIGIN(BYTE,BYTE(11))
FND(BYTE) DOS$CONFIGD(BYTE),DOS$CONIGS(BYTE)
```

DOS$STATUS(*unit*,*status*)-->*error*

The **DOS$STATUS** function performs a disk status request directly through the SIO bus (not the same as the STATUS request of ATARI BASIC) for the specified ***unit*** (**1-8**). The four ***status*** bytes returned from the device are put in the array passed as the second parameter. The value returned is the error status of the comand. An ***error*** value of 1 indicates the operation went OK. Other values are the usual error codes one might get in the 128 to 255 range. This and the subsequent functions in the package are usually called using the BASIC **ERROR** statement, which see.

DOS$READ(*unit*,*address*,*sector*,*density*)-->*error*

The **DOS$READ** function does a sector read. The ***unit*** is the disk number (**1**-*8*), the ***address*** is the address of the buffer or holding the sector, the ***sector*** is the sector number (**1-720**), and the ***density*** is the density of the drive (**1** for single, *2* for double).

DOS$WRITE(*unit*,*address*,*sector*,*density*)--> *error*

The **DOS$WRITE** does a sector write. Its parameters are the same as **DOS$READ**. It is effected by the **VERIFY** and **NOVERIFY** extrinsic commands.

DOS$CONFIGIN(*unit*,*table*)-->*error*

The **DOS$CONFIGIN** reads a config block from a percom compatible disk drive. '***unit***' specifies which drive unit. '***table***' is a table of twelve bytes showing the configuration. They have the following meanings:

byte #	description
0	Number of tracks

1	Step rate
2,3	# sectors per track
4	# sides -1
5	density (0=single, 4=double)
6,7	# bytes per sector
8	drive selected?
9	serial rate value
10,11	reserved

(two byte values are high-order-byte-first)

DOS$CONFIGD(*unit*)-->*error*

DOS$CONFIGD configures the specified disk ***unit*** to be Double density.

DOS$CONFIGS(*unit*)-->*error*

DOS$CONFIGS configures the specified disk ***unit*** to be Single density.

$ LOAD MATH

BASIC declarations:

```
FNC(FLOAT) MODFLT(FLOAT,FLOAT)
FNC(FLOAT) SIN(FLOAT),COS(FLOAT),TAN(FLOAT)
FNC(FLOAT) SQR(FLOAT)
FNC(FLOAT) ATN(FLOAT),ATN2(FLOAT,FLOAT)
FNC(FLOAT) ASN(FLOAT),ACS(FLOAT)
FNC(FLOAT) SINH(FLOAT),COSH(FLOAT),TANH(FLOAT)
FNC(FLOAT) ASNH(FLOAT),ACSH(FLOAT),ATN(FLOAT)
PROC RAD,DEG
FNC(FLOAT) RND,RAND(FLOAT,FLOAT)
FNC(BINARY) RND2,RAND2(BINARY,BINARY)
FNC(BYTE) RND1,RND2(BYTE,BYTE)
FNC(FLOAT) ROUND(FLOAT,BYTE)
```

The **MATH** package provides several of the usual floating point functions not provided directly in the OS. The random functions are also provided in the **GAME** package. Since most of these functions are obvious and familiar, most will be given only short, even one-word, descriptions

MODFLT(*float1*,*float2*)-->*float3*

The **MODFLT** divides ***float1*** by ***float2*** and returns the remainder.

SIN(*float*)-->*float*

Sine.

COS(*float*)-->*float*

Cosine

TAN(*float*)-->*float*

Tangent

SQR(*float*)-->*float*

Square root

ASN(*float*)-->*float*

Arc sine

ACS(*float*)-->*float*

Arc cosine

ATN(*float*)-->*float*

Arc tangent

ATN2(*float1*,*float2*)-->*float3*

ATN2 returns the following values for the following signs of float1 and float2:

float1	*float2*	**ATN2**
all	>0	**ATN(*float1*/*float2*)**
>0	=0	**pi/2**
<0	=0	**-pi/2**
>=0	<0	**pi+ATN(*float1*/*float2*)**
<0	<0	**-pi+ATN(*float1*/*float2*)**
=0	=0	**ERROR 5**

In essence, this function, given (x,y) coordinates, returns the angle of polar coordinates.

RAD
DEG

These procedures cause the trig functions to use degrees or radians and return the same. The default is radians.

SINH(*float*)-->*float*
COSH(*float*)-->float
TANH(*float*)-->*float*
ASNH(*float*)-->*float*
ACSH(*float*)-->*float*
ATNH(*float*)-->*float*

These are the hyperbolic functions

RND1-->*byte*

RND1 returns a random byte (0-255)

RAND1(*byte1*,*byte2*)-->*byte3*

RAND1 returns a random byte in the range ***byte1*** to ***byte2*** inclusive.

RND2-->*binary*

RND2 returns a random binary value (0-65535)

RAND2(*binary1*,*binary2*)-->*binary3*

RAND2 returns a random binary value in the range ***binary1*** to ***binary2***.

RND-->*float*

RND returns a random floating point number in the range 0 to 1.

RAND(*float1*,*float2*)-->*float3*

RAND returns a random floating point number in the range ***float1*** to ***float2***.

ROUND(*float1*,*byte*)-->*float2*

ROUND returns ***float1*** rounded to ***byte*** places after the decimal point. 255, 254,... will be considered -1, -2, ... That is, it will be rounded off on the left of the decimal point.

$ LOAD RS232

BASIC declarations:

```
PROC OS$CIOV(BYTE,BYTE,BYTE,BYTE,STRING(*))
PROC R$STARTCON(BYTE,BINARY,BINARY)
FNC(BINARY) R$BUFFIN(BYTE)
PROC R$CONTROL(BYTE,BYTE,STRING(*))
PROC R$BAUD(BYTE,BYTE,BYTE,STRING(*))
PROC R$TRANSLATE(BYTE,BYTE,BYTE,STRING(*))
PROC R$FORCE
```

When the **RS232** module is loaded, the RS232 (ATARI 850) interface module must be on. It downloads the actual dre summarized below, but are not a replacement for the ATARI 850 Reference Manual. This module may be **KILL**ed. When it is **KILL**ed, the hooks RS232 makes into the device table and the reset vector are undone. All operations set the **ERR** function value to one.

OS$CIOV(*channel*,*command*,*aux1*,*aux2*,*device*)

The **OS$CIOV** procedure is the equivelant of ATARI BASIC's **XIO** statement. Its primary use is by the following subprograms.

R$BUFFIN(*channel*)-->*input_buffer_size*

The **R$BUFFIN** function performs a OS status request. It returns the current size of the input circular buffer. See Appendix 4 of the ATARI 850 manual.

R$STARTCON(*channel*,*buffer address*,*buffer size*)

The **R$STARTCON** procedure activates concurrent i/o on an open RS232 port. This is discussed in Appendix 8 of the ATARI 850 manual. If buffer address is 0, the internal 32-byte buffer is used. Otherwise, you may provide your own buffer of larger size (which is a good idea for faster transfer rates).

R$CONTROL(*channel*,*aux1*,*device*)

The **R$CONTROL** procedure does the **XIO 34** operation documented in Appendix 7 of the ATARI 850 manual. Briefly, the following values may be added to ***aux1***:

0	No change to DTR	0	No change to RTS	0	No change to XMIT
128	Turn DTR off	32	Turn RTS off	2	Set XMT to space(0)
192	Turn DTR on	48	Turn RTS on	3	Set XMT to mark(1)

R$BAUD(*channel*,*aux1*,*aux2*,*device*)

The **R$BAUD** procedure does the **XIO 36** operation documented in Appendix 5 of the ATARI 850 manual. The following values may be added to ***aux1***:

BAUD			
0	300	8	300
1	45.5	9	600
2	50	10	1200
3	56.875	11	1800
4	75	12	2400
5	110	13	4800
6	134.5	14	9600
7	150	15	9600
WORD SIZE			
0	8 bits	16	7 bits

3	6 bits	48	5 bits
STOP BITS			
0	1 bit	128	2 bits

The following values may be added to ***aux2*** to enable monitoring of DSR,CTS, CRX:

1	monitor CRX
2	monitor CTS
4	monitor DSR

R$TRANSLATE(*channel,aux1,aux2,device*)

The **R$TRANSLATE** procedure does the **XIO 38** operation documented in Appendix 6 of the ATARI 850 manual. The following values may be added to ***aux1***:

0	Light ATASCII/ASCII translation
16	Heavy translation
32	No translation
0	Ignore input parity
4	ODD input parity
8	Even input parity
12	SPACE input parity
0	Do not modify parity bit
1	ODD output parity
2	EVEN output parity
3	MARK output parity
0	Do not append LF to CR
64	Append LF to CR

Aux2 is the heavy translation character.

R$FORCE(*channel*)

The **R$FORCE** procedure does the **XIO 32** operation documented in pp. 27ff of the ATARI 850 manual. It forces early transmission of the output buffer if you are using block output.

$ LOAD SOUND

????????????????????????????????

IV. Miscellaneous

A. GRAPHICS modes

The following table of modes is common to all languages as they are implemented directly in the Operation System.

Pixels				
mode	type	across	down	pixels
0	text	40	24	1(2)
1	text	20	24(20)	5
2	text	20	12(10)	5
3	graph	40	24(20)	4
4	graph	80	48(40)	2
5	graph	80	48(40)	4
6	graph	160	96(80)	2
7	graph	160	96(80)	4
8	graph	320	192(160)	1(2)
9	graph	80	192	1(16)
10	graph	80	192	9
11	graph	80	192	16(1)
12	text*	40	24(20)	5
13	text*	40	12(10)	5
14	graph	160	192(160)	2
15	graph	160	192(160)	4
* resolution 4x8 multicolor pixes				

The above mode numbers specify modes (with the four line text window in parenthesis, if applicable). Add 16 to suppress the text window. Add 32 to suppress the screen clear. Add 48 to do both.

Hues	Color
0	Gray
1	Light Orange(Gold)
2	Orange
3	Red-Orange
4	Pink
5	Purple
6	Purple-Blue
7	Blue 1
8	Blue 2
9	Light Blue
10	Turquoise
11	Green-Blue
12	Green
13	Yellow-Green
14	Orange-Green
15	Light Orange

Defaults			
Register	Hue	Luminance	Description
PF0	2	8	Orange
PF1	12	10	Green
PF2	9	4	Dark Blue
PF3	4	6	Pink/Red
PF4	0	0	Black

Use of Registers		

Modes	Register	Color
0,8	PF1	character,color 1
	PF2	background
	PF4	border
1,2,12,13	PF0	character
	PF1	character
	PF2	character
	PF3	character
	PF4	background,border
4,6,14	PF0	color 1
	PF4	color 0,border
3,5,7,15	PF0	color 1
	PF1	color 2
	PF2	color 3
	PF4	color 4,border
9	PF1	colors=luminance
10	PM0	color 0
	PM1	color 1
	PM2	color 2
	PM3	color 3
	PF0	color 4
	PF1	color 5
	PF2	color 6
	PF3	color 7
	PF4	color 8
11	PF2	color=hue
PMn refers to Player-Missile registers; PFn refers to playfield registers.		

B. Devices

C:		cassette
D1:	D5:	disk drives
D2:	D6:	
D3:	D7:	
D4:	D8:	
P1:	P5:	printers
P2:	P6:	
P3:	P7:	
P4:	P8:	
R1:	R3:	RS-232 ports
R2:	R4:	
S:		graphics screen
K:		keyboard
E:		screen editor

C. Error Codes

0,1	all's well that ends well
The following error codes are unique to the pcode of the RTL:	
2	floating point add overflow
3	floating point subtract overflow
4	floating point multiply overflow
5	floating point division overflow
6	too high dimension number in BOUND function

7	conversion error: floating point to integer
8	conversion error: string to floating point
9	conversion error: integer to byte
10	e to a power overflow
11	10 to a power overflow
12	base e log of negative number
13	base 10 log of negative number
14	negative exponent in raise to a power
15	multiply overflow in raise to a power
16	e to a power overflow in raise to a power
63	this error is returned by many provided modules
64-127	You should use these when creating your own error conditions.
128-255	are the usual systems errors
128	break abort
129	channel already open
130	nonexistant device
131	channel write only
132	illegal handler command
133	device/file not open
134	bad channel number
135	channel read only
136	end of file
137	truncated record
138	device timeout
139	device NAK
140	serial frame error
141	cursor out of range
142	serial bus overrun
143	checksum error
144	device done error

145	illegal screen mode
146	function not implemented
147	insufficient RAM
150	port already open (RS-232)
151	concurrent mode not enabled
152	illegal user supplied buffer
153	active concurrent mode I/O error
154	concurrent mode I/O not active
160	drive number error
161	too many files open
162	disk full
163	unrecoverable system I/O error
164	file number mismatch
165	file name error
166	POINT data length error
167	file locked
168	device command invalid
169	directory full
170	file no found
171	POINT invalid
172	illegal append
173	bad sectors at format time
176	incompatable format

D. Converting to double density

You will need to **LOAD**:

COMMAND, **IO**, **COPY**, **CONFIG**, **SDCOPY**, **PUTDOS**

first. You should also read their documentation first.

One disk drive

1. insert blank disk in drive one
2. **$ CONFIG 1 D**
3. **$ FORMAT 1**
4. **$ PUTDOS**
5. insert system disk in drive one
6. **$ SDCOPY *.* *.* /QUERY**
7. on the **SDCOPY**, answer **N** (no) to **DOS.SYS**. Never **SDCOPY DOS.SYS**!

Two or more disk drives

1. insert system disk in drive one
2. insert a blank disk in drive two
3. **$ CONFIG 1 S**
4. **$ CONFIG 2 D**
5. the above two commands insure that the disks have the proper densities
6. **$ FORMAT 2**
7. **$ PUTDOS** (specify disk unit **2** when prompted)
8. **$ COPY *.* *.* /QUERY**
9. on the **COPY**, answer **N** (no) to **DOS.SYS**. Never **COPY DOS.SYS**!

D. Vocabulary

argument (for CP commands)
: a piece of information provided to a command the meaning of which is dependent on its order with respect to the other arguments. **Arguments** are usually (but not always) prompted for when omitted.

argument (of functions)
: see **parameter**

batch
: a set of **CP** commands in a file that **CP** can interpret.

channel
: the same as an **IOCB**. It is, to the programmer, an integer between zero and seven inclusive used to refer to a file.

compiler
: a program that converts a high level language into a lower level one (such as a pcode or machine language).

Control Program
: the program that controls the computer for the user.

CP
: same as **Control Program**.

default device
the device used in a **file specification** (which see) when a device is not expicitly specified.

extender
see **file specification**.

external entry point
a subprogram in a module that may be called by other modules.

external reference
a reference by a module to a subprogram contained in some other module.

extrinsic command
a module that may be ran. It must be **LOAD**ed into RAM first. Same as **Utility**.

filename
see file specification

file specification (**filespec**)
the specification of a file or device. It consists of three fields: device_letter[digit]:filename [up to eight letters][.extender [up to three letters]]

format
to prepare a disk for storing files.

function
a **subprogram** (which see) that can return a value for use in an expression.

intrinsic command
a command that is done by code in **CP** itself.

IOCB
properly, a sixteen byte block that controls I/O. To the programmer it is a number from zero to seven used to refer to a file. Same as **channel**.

link
to resolve an **external reference** by pointing it to an **external entry point**.

machine language
the series of bits (ones and zeros) directly understood by the hardware of a computer.

option
syntactical unit in **CP** like ***/keyword=value***, ***/keyword***, and **/NO_*keyword***. They are optional, have default values, and are not prompted for when omitted.

package
a module whose entire contents (or nearly so) are subprograms used as external entry points. They allow several other modules to share the same copy of subprograms as well as allow the subprograms in the package use information hidden (protected) from those other modules. See module, utility.

parameters & **arguments**
in the following segment:

```
CALL XYS(A,B+5)
DEF XYZ(M,N)
```

A and **B+5** are **arguments** being passed to **XYZ**. **M** and **N** are the parameters used by **XYZ**. See the language manuals.

pcode
: a low level language that is interpreted by machine language. It is more primative and faster than the tokens used by BASIC interpreters.

positional argument
: see **argument** (for CP).

procedure
: a **subprogram** that does not return a value for use in an expression.

prompt
: a message to the user on the screen telling him to type something in. As a verb, to give a user such a message.

RTL (**Run Time Library**)
: Library of machine language subroutines that interpret the JLS pcode.

stack
: a structure for storing data into which data may be added and removed from only one end, called the top of the stack. It is analogous to a stack of dishes.

subprogram
: a programming unit that can be called by another programming unit and will return to that unit. It has a name, as opposed to a subroutine which is called with its line number in BASIC. Subprograms can also have parameters, local variables, and local subprograms. There are two types of subprograms. FUNCTIONs can be used in expressions since they return a value. PROCEDUREs are used like program statements: they perform an operation, but return no value.

syntax
: the rules of grammar of a language. Syntax specifies valid arrangements of characters and words, as opposed to semantics which specify what those combinations mean.

terminal
: see **virtual terminal**

utility
: a module whose primary purpose is to ba run as an extrinsic command.

virtual terminal (**VT**)
: there is a maxim: "if its there and you can see it, its real; if its there and you can't see it, its transparent; if its not there but you can see it, its virtual; if its not there and you can't see it, its been deleted." A terminal is a device that transmits over a communications line what is typed on its keyboard, and displays on its screen what it receives over that line. A virtual terminal probram makes your ATARI look like a real terminal.

wildcard
: a character in a filespec (filename or extender) that can substitute for any other valid character(s).

V. BASIC

A. The Command Line

$ BASIC ***sourcefile***
/OBJECT=*.OBJ
/LIST=E:
/ERROR=E:
/BASE=0

(requires **COMMAND**)

Wildcards used in **/OBJECT**'s filename are replaced with the corresponding characters from the source filename.

/OBJECT=the file the object module is sent to
/LIST=the file the listing is sent to
/ERROR=the file error messages are sent to
/BASE=the base address of the object module. **0** is for relocatable modules. **49152** is for the CP.

B. Writing a program

As is common in most BASICs, every line of the program must have a unique number that is in ascending order (that is, the line numbers determine the order of the lines). The maximum line number permitted is **32767**. The **EDIT** module should be used to edit a BASIC program. As in ATARI BASIC, the maximum linesize is 120 characters, and more than one statement may be put on one line by seperating them with colons (:). In examples given below, line numbers are entirely arbitrary. The form of a typical, simple program is given below.

```
10 REM this is an example
20 DEF MAIN
30   DECLARE
40     <variable declarations>
50   ENDDECLARE
60   <program statements>
70 END
80 this is ignored
```

All variables must be declared. Declaring a variable is necessary because of the number of types, scope considerations (see C. Writing Subprogams), and the fact that this is a one pass compiler. Lowercase letters are not permitted in keywords or identifiers. The first character of a variable must be a capital letter. The subsequent letters, if any, may be any of the capital letters, digits (**0-9**), **#**, **!**, **%**, **$**, and **_**. Variables are limited to **15** characters. There are four types of variables: **BYTE**, **BINARY**, **FLOAT**, and **STRING**. **BYTE** can hold integers of the range **0** to **255**. **BINARY** handles the integer range **0** to **65535**. **FLOAT** is the same as ATARI BASIC: **-9.99999999E-98** to **9.99999999E+98**. **STRING** can be up to **253** characters. Variables may have up to three dimensions. The following is an example declaration:

```
30 DECLARE:BYTE A,B(5)
40   BINARY C,D(5,6,7)
50   FLOAT E
```

```
60   STRING(31) F:STRING(15) G(99)
70 ENDDECLARE
```

A is a simple **BYTE** variable. **B** is an array of six **BYTE**s. Note that the lowest subscript is zero. **C** is a simple **BINARY** variable, and **D** is a three-dimensional array of **BINARY** values. **E** is a floating point variable. F is a **STRING** variable whose maximum length is **31** characters. **G** is an array of **100 STRING**s whose maximum length is **15** characters each. The memory requirements of G are (15 characters+ 1 length byte) times 100 elements = 1600 bytes. **BYTE**, **BINARY**, and **FLOAT** elements take one, two, and six bytes each respectively.

Numeric constants have type too. You should use the proper type in a given context to avoid conversion operations, which will slow down program execution. The following are some examples of constants of each type:

```
BYTE: &5 &0 &255

BINARY: 5 0 255 743 65535

FLOAT: 5. 0. 255.0 65535. 2.3765 2.4E-5 .2 2E10
```

Generally, if the number has, or starts with a period, or has an **E** in it, it is a floating point constant. If it consists of only digits, it is a **BINARY** constant. If it starts with an ampersand (**&**) followed by digits only, it is a **BYTE** constant.

Expressions are the next level of complexity up from variables and constants on the way to constructing statements and whole programs. There are three other elementary units used in constructing an expression: functions, unary operators, and binary operators.

The following is the order of precidence table of these operations:

functions
+ - (unary)
^ (raise to a power)
***** **/**
+ -
AND
OR
EOR
; (Concatenate strings)

Operations of equal precidence are done from left to right.

Type coercions are performed for mismatched types (eg, binary+byte). The rules are somewhat unusual so care should be taken. It is advised that you use explicit conversion functions to avoid errors anyway. In fact, some languages permit no coercions at all.

functions

all arguments of functions are converted to the expected types. Some built-in functions might make exceptions.

Unary operators (+ and -)

do not covert numbers, but if used on a **STRING**, the string is converted to **FLOAT** first. The value returned is of the same numeric type was the operation was performed on.

Binary operators

the value returned is the same as both of the values acted upon. Usually only the right hand expression is converted. The following shows the type returned for each operator and type of its left and right hand expressions. The left and right hand expressions are coerced to the return type.

any* ^ *any --> **FLOAT**

BYTE or **BINARY *** or **/** ***any*** --> **BINARY**

FLOAT * or **/** ***any*** --> **FLOAT**

STRING * or **/** ***any*** --> **FLOAT**

BYTE + or **-** ***any*** --> **BYTE**

BINARY + or **-** ***any*** --> **BINARY**

FLOAT + or **-** ***any*** --> **FLOAT**

STRING + or **-** ***any*** --> **FLOAT**

BYTE AND or **OR** or **EOR** ***any*** --> **BYTE**

not **BYTE AND** or **OR** or **EOR** ***any*** --> **BINARY**

any* ; *any --> **STRING**

The **^**, *****, **/**, **+**, and **-** do the same operations they do in ATARI BASIC. **AND**, **OR**, and **EOR** perform bitwise ands, ors, and exclusive ors on byte and binary values. See also the **NOT** function for bitwise operations. The semi-colon concatenates (combines) two strings.

On the level of the statement, the main thing to beware of is delimiting between lexical units. The following illustrates this:

```
FORA=BTO7

CLOSE#5
```

The first would be seen as a simple assignment statement. **FOR** must be seperated from the **A** with one or more spaces. Neither **A** nor **B** need be seperated from the equal (**=**) sign however. It cannot be part of a symbol name and no other lexical unit starts with equal followed by a letter. **B** and **7** must be seperated from **TO** however since **BTO7**, **BTO**, and **TO7** are all valid variable names. The problem in the second one is more subtle. Though you should avoid such a name, **CLOSE#5** is a valid variable name. **CLOSE# 5** would solve the problem. So would **CLOSE#&5**, since **&** cannot be part of a variable name, the compiler can see the start of another lexical unit.

On the program level, the first non-**REM**ark statement must be **DEF MAIN** (or later we will see **DEF PACKAGE**). The next statements after that, if the program uses variables, are the **DECLARE** ... **ENDDECLARE** statements. After that come the program statements and, finally, the **END** statement. Everything after the **END** statement is ignored.

C. Using Subprograms

Like all variables, subprograms must also be declared in the DECLARE block at the head of a program. Subprogram names have the same size and character limitations as variable names. There are four types of subprograms:

FNC	function
RECFNC	recursive function
PROC	procedure
RECPROC	recursive procedure

A function is a subprogram that returns a value and can be used like any built-in function. A procedure is like a statement in that it does not return a value. It is called using the **CALL** statement. A recursive subprogram is one that can call itself directly or by calling a subprogram that calls it or by calling a subprogram that calls a subprogram that calls it ....

The following annotated examples will illustrate how to **DECLARE** and use procedures:

```
10 DEF MAIN
20   DECLARE
30     BINARY A,B,V
40     FLOAT D,E,F
50     STRING(16) G:STRING(128) H
60     PROC HEADER
70     PROC TITLE(STRING(128))
80     PROC CHANGE(STRING(*))
90   ENDDECLARE
```

Note that in the above declarations, all the variables are **DECLARE**d before the procedures. This is advised to avoid a scope problem that will be explained later.

```
100   CALL HEADER
110   CALL TITLE(G)
120   CALL TITLE("JLS manual")
130   CALL CHANGE(H)
140   CALL CHANGE(G)
150   CALL CHANGE("TEST")
1000  DEF HEADER
1010    PRINT:PRINT "HELLO":PRINT
1020    RET
1030  ENDDEF
```

All subprograms should be **DEF**ined after all the main program statements. If the sequence of execution ever "runs into" a **DEF** statement (as in line **150** to **1000** above), a **END** or **STOP** occurs. The **CALL** in line **100** is similar to a **GOSUB**, but instead of a line number, it has a procedure name after it. Lines **1000-1030** contain a subprogram definition. Note it starts with a **DEF** and ends with a **ENDDEF**; this is necessary. Also necessary are one or more **RET** statements. **RET** is to **CALL** what **RETURN** is to **GOSUB**. It signals a **RET**urn to the statement after the **CALL** that invoked that procedure.

```
2000  DEF TITLE(V)
2010    IF V="":PRINT "DEFAULT":RET
2020    ELSE:PRINT V:RET
2030    ENDIF:ENDDEF
```

This example illustrates another valuable feature of subprograms: parameter passing. Note that line **110** passes a variable to the subprogram. Line **120** passes it a **STRING** constant. Line **2000**, in addition to heading the definition, gives a name to the parameter whose type was all that was given in the declaration on line **70** (the * will be discussed later). The parameter is just a regular variable, but the name may be used only between the **DEF** and **ENDDEF** of **TITLE**. **TITLE** can use all variables declared in the main program (ie, **A**,**B**,**D**,**E**,**F**,**G**,**H**) because it is between **DEF MAIN** and END, except one: the use of **V** as **TITLE**'s parameter name hides the existence of **V** in the main program. Consider the following implementation of **TITLE**:

```
2000  DEF TITLE(V)
2010    DECLARE:BINARY F,Z:ENDDECLARE
2020    ...
2030  ENDDEF
```

This one declares some more variables. These are also local to **TITLE**. The compiler will inform you that **Z** is not declared if you try to use it outside of the **TITLE**. Note that there is an **F** declared in the **MAIN** program too. Inside **TITLE**, **MAIN**'s **F** is hidden by **TITLE**'s **F**. Generally, one should avoid having a subprogram use variables declared outside it. By having a subprogram get all its information from parameters and using only local variables to do its manipulations, writing subprograms that can be used by other programs that you or someone else might write is facilitated. This modular design method also reduces the chance of making programming errors.

```
3000 DEF CHANGE(V):REM note that V is used again
3010   V="(";V;")"
3015   REM remember semicolon (;) is the concatenation operator
3020 RET:ENDDEF
```

This example illustrates a more subtle, but extremely powerful, feature: pass-by-reference. Until now, I might have implied that parameter passing is accomplished with an assignment statement (**V=*whatever***). This is not so. Note the declarations of **H** and the parameter **V**. Both are of type **STRING(128)**. In the case of line **130**, **V** becomes another name for **H**. When you change **V**, **H** also is change since they are the same. The above procedure adds parentheses to the beginning and end of the parameter. The situation is quite different in line **140**. That **G** is declared **STRING(16)**. Since this is not **STRING(128)**, a variable of that type is allocated on the stack, **G** is copied to it, and that variable is passed to **V**. Hence when **V** is changed, **G** is not. The same thing happens in line **150** where an expression is passed.

Now observe how **CHANGE** is declared. This would cause both **G** and **H** to be passed by reference. The * acts as a "wildcard" similar to its function in filenames. Obviously, expressions are still passed by value anyway.

It is possible to pass entire arrays by reference also. Consider the following example:

```
110 DEF MAIN
120   DECLARE
130     FLOAT A(99),B(100)
140     STRING(8) C(99),D(200)
145     STRING(9) E(99)
150     PROC CLEARFLT(FLOAT(99))
160     PROC CLEARSTR(STRING(8)(*))
170   ENDDECLARE
250   CALL CLEARFLT(A)
260   CALL CLEARSTR(C)
270   CALL CLEARSTR(D)
275   CALL CLEARSTR(E)
280   CALL CLEARFLT(B)
```

The procedure **CLEARFLT** takes a one dimensional array of floating point numbers that has **100** elements as its parameter. Hence line **250** is valid. Line **280** would result in an error becuase array **B** has 101 elements -- the wrong number. If the bound size of the array is specified as an * in the procedure declaration, any size on that dimension can be passed. Note this in line **160**. Note also the two sets of parenthesis in it: **PROC CLEARSTR((8)(*))**. The **(8)** specifies the maximum string size. It too can be an *. Note that in line **275** an error would occur: **E** has maximum element size of 9. Now, consider the following declaration:

```
PROC DOSOMETHING(STRING(80)(10,*,5),BINARY(5,*))
```

This procedure takes two parameters. The first is a three dimensional array of strings whose maximum length is **80**. The first and third dimensions have upper bounds of **10** and **5** respectively while the second dimension may be of any size. The second parameter is a two dimensional array of **BINARY** numbers. The first dimension must have an upper bound of **5**, but the second one might be any size.

Functions are defined much like procedures, but they return a value and are used like built-in functions in expressions. Consider the following program:

```
10   DEF MAIN
20     DECLARE
30       FLOAT TEMP
40       FNC(FLOAT) F_TO_C(FLOAT)
50     ENDDECLARE
60     PRINT "Enter temperature in Fahrenheit: ";:INPUT TEMP
70     PRINT F_TO_C(TEMP);" degrees Celcius"
80     PRINT:GOTO 60
1000   DEF F_TO_C(T)
1010     RET (T-32.0)*9.0/5.0
1020   ENDDEF
9999 END
```

The format **FNC(FLOAT)** specifies non-recursive functions that return a floating point value. The rest of the **FNC** statement is the same as for **PROC**. There is no difference in the **DEF** and **ENDDEF** of **FNC** either. The difference is in the **RET** statement. Here it has an expression after it containing the value to be returned. Parameters are passed to, and can be modified by, functions just like in procedures. When a function returns a **STRING** value, you do not specify a maximum length in the declaration:

```
FNC(STRING) CHANGE(STRING)
```

For an example of recursion, the classical factorial example will be used. In mathematics, the notation n! (read as n factorial) is the product of all integers from 1 to n inclusive. We can define this using recursion:

```
n!= if n=0, 1
    if n>0, n(n-1)!
0!=1 by definition.
```

Factorial of all other positive integers is given using factorial of one less than each integer. Here is the code for implementing this function:

```
20       RECFNC(FLOAT) FACTORIAL(BINARY)
1000   DEF FACTORIAL(N)
1010     DECLARE:FLOAT T:ENDDECLARE
1020     IF N=0:RET 1.0
1030     ELSE
1040       T=FACTORIAL(N-1)
1050       RET FLT(N)*T
1060     ENDIF
1070   ENDDEF
```

This could have been written to not use **T**, but **T** is needed to make a point. Each time **FACTORIAL** calls itself (in line **1040**), a new **N** and **T** are created. When **RET** is done, they are destoyed and the old **N** and **T** become visible again. One way of viewing a recursive subprogram is to imagine each definition has another copy of the subprogram declared and defined in it which has another copy declared and defined in it and so on for an infinite regression. Actually, the regression is not infinite because you will eventually run out of stack space.

Now obviously, it would be much easier to implement factorial without recursion that would run much faster as well. There are algorithms, however, that are much easier to implement using recursion: for example, the Towers of Hanoi problem, the BASIC compiler, and Quicksort. There is also another reason why one might declare procedures as recursive, even if they are not used so. When a subprogram is non-recursive, the space for local variables is statically (permanently) allocated. That is, allocation is done at **LOAD** time. If subprograms have large variable space requirements, memory is short, and a subprogram's variable's values need not be kept from one call to another, then you maybe should declare the subprogram as recursive. This causes it to allocate memory for its variables when **CALL**ed and deallocate it when it **RET**urns so that other subprograms can used the same memory. However, this efficiency of space is not without a price-tag. Access to dynamically allocated variables and recursive **CALL**s and **RET**urns take more time (about 10% longer, more or less).

For these reasons, several of the subroutines provided in packages are recursive even though they are not used recursively.

D. Using and Making External Subroutines (Packages)

The text of a subprogram that is used by a program (or even another subprogram) does not have to actually be included with the program or subprogram during compilation. It could have been compiled separately. To use a subprogram that has been compiled in a separate module (package), all you need do is **DECLARE** it, but never **DEF**ine it. When your program is **LOAD**ed, all such unresolved references are resolved using packages that are already **LOAD**ed in. The proper **DECLARE**ations to use are documented in Chapter 3-B.

Any program can let the CP use some or all of its subprograms to resolve unresolved references of other programs. This is done by using **DEF PACKAGE** instead of **DEF MAIN**. The **PRIVATE** statement is used in the **DECLARE** block to restrict which are used. The following is the suggested order of declarations:

```
10 DEF PACKAGE
20   DECLARE
30     <variables>
40     <references to external routines>
50     <subprograms that may be used by other modules>
60   PRIVATE
70     <subprograms used internally only>
80   ENDDECLARE
```

All subprograms declared within other subprograms are always private. All subprograms declared before **PRIVATE** can be used by other programs. Those declared after it are kept private to the program. Declarations of subprograms that are not **DEF**ined in the program may actually come either before or after the **PRIVATE** statement, but they should be grouped together for program readability purposes.

One thing to remember is that subprograms that are contained in other modules cannot access variables in the module that is calling them. They might access variables declared globally in the module that defines them. They may then share data and subprograms that are not visible to your program.

Commonly, modules that provide subroutines have no code of their own. For example, if you issue **COMMAND** as a command, nothing will happen. There is also another use of **DEF PACKAGE** that does not have to provide entry points that can be used for other modules. Consider the following segment:

```
10 DEF PACKAGE
20   DECLARE
30     <variables>
40     RECPROC ATTACH,DETACH
50     <external references>
60     <external entry points>
70   PRIVATE
80     <private subprograms>
90   ENDDECLARE
```

The CP will **CALL ATTACH** when the module is **LOADED**, and will **CALL DETACH** when the module is **KILL**ed. An example of how this has been used is the RS232 module. When the RS232 is **LOADED**, **ATTACH** downloads the RS-232C driver from the ATARI 850 interface module and adds the driver to the system device tables and sets the reinitialization vector for RESETs. **DETACH** removes the **R** entry from the device table and removes the RS-232 initialization vector from the RESET vector, restoring the old value.

(The following sections summarize all of the statements of JLS BASIC. The syntax notations specify the types of each argument in lower case letters. The notations ***byt***, ***bin***, ***flt***, and ***str*** mean **BYTE**, **BINARY**, **FLOAT**, and **STRING** operands respectively. When ***varbyt***, ***varbin***, ***varstr***, ***varflt*** appear, a variable of the specified type is required (that variable will probably have its value changed). Each argument is also numbered, starting with zero, for reference. Anything in **[**brackets**]** is optional).

E. Control Statement Summary

ELSE -- see IF

ELSEIF -- see IF

ENDIF -- see IF

ENDWHILE -- see WHILE

ERROR *byt0*

The **ERROR** statement simulates error number ***byt0***. There is a special case: **ERROR &1** just clears the **ERR** function without generating an error. See **ERR** function.

EXEC *bin0*

The **EXEC** statement does a JSR to a machine language subroutine. The average user will probably never use this directly. Some of the provided packages use this. Most things you might want to use this statement for are already provided in those packages.

FOR *var0=num1* TO *num2* [STEP *num3*]
  statements
NEXT *var0*

The **FOR**...**NEXT** statements provide looping control. ***Var0*** may be **BYTE**, **BINARY**, or **FLOAT**. ***Num1***, ***num2***, and ***num3*** should be the same type as ***var0***. The statement is very similar to ATARI BASIC's. ***Num3*** will default to one (**&1**, **1**, or **1.0**) if the **STEP** clause is omitted. The body of statements will always be executed at least once. **Num3** may be negative to count down from **num1** to **num2**. There are some differences.

There may be only one **NEXT** statement for each **FOR** statement. The **FOR**...**NEXT**s must be both physically and logically nested. You may freely **GOTO** out of the block of statements, even back to a previous **FOR** statement. For byte and binary types, ***num1*** and ***num2*** are always considered positive. When ***var0*** is byte the following obtains: **&128** to **&255** are considered -128 to -1 respectively for ***num3*** only; ***num2*** should never be **&255** when you are counting up, nor **&0** when counting down.

Similarly, when ***var0*** is binary: **32768** to **65535** are considered -32768 to -1 for ***num3*** only; ***num2*** should never by **65535** when counting up, nor **0** when counting down. The reason for the restriction on ***num2*** is this. If **var0** is **BYTE**, ***num2*** is **&255**, ***num3*** is **&1**, ***var0*** has reached **&255**, and **NEXT var0** is being executed, then **&1** is added to ***var0*** (**&255+&1**) which results in **&0**. Since **&0<&255**, **NEXT** loops back to **FOR**. An infinite loop obtains!

GOSUB *line_no*

RETURN

GOSUB transfers control to the specified ***line_no*** (***line_no*** is specified using a **BINARY** constant). After **GOSUB**, when **RETURN** is encountered, control returns to the statement after the **GOSUB**. **GOSUB**s may be freely nested. The levels of nesting however should be confined to about 60 or so. **GOSUB** uses the hardware stack to store the return address.

GOTO *line_no*

The **GOTO** statement transfers control to the specified line number.

IF *condition* THEN *line_no*

The **IF** statement is a conditional **GOTO**. The ***condition*** is a **BYTE** expression. If the ***condition*** is **&255**, a **GOTO *line_no*** is executed. If ***condition*** is **&0**, execution continues with the next statement. Other values of ***condition*** should be avoided. Remember, the relational operators return **&255** for true and **&0** for false.

IF *condition0*
    statements0
[ELSEIF *condition1*
    statements1
[ELSEIF *condition2*
    ***statements2*...]]**
[ELSE
    ***statements3*]**
ENDIF

This is another syntax of the **IF** statement. Only one of of the groups of ***statement***s will be executed. If ***condition0*** is true (**&255**) then only ***statements0*** will be done. If false, ***condition1*** is evaluated. If ***condition1*** is true, only ***statements1*** will be executed. If false, ***condition2*** is evaluated. If ***condition2*** is true, only ***statements2*** will be executed. There may be any number, even zero, of such **ELSEIF** statements. They may span may lines. In fact, the statements in three such complexes of **IF**...**ELSEIF**...**ELSE**...**ENDIF**s make up most of the JLS BASIC compiler. If all of the conditions evaluate to false, ***statements3*** after the **ELSE** statement are done. The **ELSE** statement is also optional. When there is no **ELSE**, it is possible that none of the statements between the **IF** and **ENDIF** will be done. The **ENDIF** is required. Some examples are given below:

```
10 IF A=B:PRINT "EQUAL":ENDIF
20 IF X<>Y:PRINT "<>":ELSE:PRINT "=":ENDIF
30 IF M<N:PRINT "<":ELSEIF M=N:PRINT "="
40 ELSE:PRINT ">":ENDIF
50 IF P=&1:PRINT "RED"
60 ELSEIF P=&2:PRINT "BLUE"
70 ELSEIF P=&3:PRINT "GREEN"
80 ELSE:PRINT "UNDEFINED"
90 ENDIF
```

NEXT -- see FOR

REPEAT
    statements0
UNTIL *condition0*

The **REPEAT**...**UNTIL** statements are another set of looping control statements. When **REPEAT** is encountered, nothing is done. ***Statements0*** are then executed. Then ***condition0*** is evaluated. If ***condition0*** is false (**&0**), ***statements0*** are done again. If true, execution goes to the next statement. Thus, ***statements0*** are executed again and again until ***condition0*** becomes true. Nesting with other **REPEAT**...**UNTIL**s and other looping control statement pairs should be both physical and logical. You may **GOTO** out of a **REPEAT**...**UNTIL** block.

RETURN -- see GOSUB

STOP

The **STOP** statement halts execution of a program, returning the user to the CP without generating an error condition.

TRAP *line_no*

The **TRAP** statement stores the specified line number and the current stack pointer away. When an error condition is subsequently raised, that line number is **GOTO**ed, and the stored stack pointer value is restored to the stack pointer. Thus unfinished expressons and subprogram calls are removed from the stack when the error occurs. There is a special case. **TRAP 0** causes error conditions to return the user to the CP with an error condition. The line number should be in the same subprogram in which **TRAP** is issued, or both should be in the main program. This is because of the effect of error trapping on the stackpointer.

UNTIL -- see REPEAT

WHILE *condition0*
    statements0
ENDWHILE

When the **WHILE** statement is encountered, ***condition0*** is evaluated. If it is true, ***statements0*** are done, else a **GOTO** is done to the statement after the **ENDWHILE** statement. When **ENDWHILE** is encountered after ***statements0*** are done, a **GOTO** back to the **WHILE** statement is done. Thus ***statements0*** are done while ***condition0*** is true. They may be done zero times. You may **GOTO** out of a **WHILE**...**ENDWHILE** block. Nesting with other **WHILE**...**ENDWHILE**s and other looping control statement pairs should be both physical and logical.

F. Defining and Declaration Statements

The below are just syntax specifications. See Sections A-D of this chapter for details on use.

DEF MAIN or DEF PACKAGE
    DECLARE

BYTE ***variable_list***
BINARY ***variable_list***
FLOAT ***variable_list***
STRING(*maxlen*) ***variable_list***
PROC ***procedure_list***
RECPROC ***procedure_list***
FNC(*return_type*) ***function_list***
RECFNC(*return_type*) ***function_list***
ENDDECLARE

variable_list :== ***variable***[(***bound1***[,***bound2***[,***bound3***]])],***variable_list***

procedure_list,***function_list*** :==***subpgmname***[(***parameter_list***)],***function_list***

return_type :== {**BYTE**,**BINARY**,**FLOAT**,**STRING**}

parameter_list :== ***type***[(***bound1***[,***bound2***[,***bound3***]])],***parameter_list***

type :== {**BYTE**,**BINARY**,**FLOAT**,**STRING(*maxlen*)**}

Note: in parameter lists, ***maxlen*** and ***bound*** may be *****.

CALL ***procedure_name*****[(*argument_list*)]**

DEF subpgmnames[(*parameter_names*)]
RET [*expression*]
ENDDEF

END -- matches DEF MAIN/PACKAGE

G. Graphics Statements

COLOR ***byt0***

The **COLOR** statement specifies which color number is to be used in subsequent **PLOT**s, **DRAWTO**s, and **FILL**s.

DRAWTO ***bin0***,***byt0***

The **DRAWTO** statement draws a straight line from the last **POSITION**, **PLOT**, **FILL**, or **DRAWTO** to the specified (x,y) location.

FILL *bin0,byt0*

The **FILL** command does a **DRAWTO**, but as it plots each point of the line, it also plots all points to the right of that point until it reaches a point that isn't already set to a non-zero color.

GRAPHICS *byt0*

The **GRAPHICS** command chooses a graphics mode for the screen. See Chapter 4 GRAPHICS MODES for various permissible values. Channel 6 is used.

PLOT *bin0,byt0*

The **PLOT** statement plots a point at the specified (x,y) coordinates ((***bin0***,***byt0***)) using the color specified in the last **COLOR** statement. See Chapter 4 GRAPHICS MODES for the valid ranges of values.

POSITION *bin0,byt0*

The **POSITION** statement positions the graphics curser at the specified (x,y) coordinates, but doesn't change the color of that point.

SETCOLOR *byt0,byt1,byt2*

The **SETCOLOR** command chooses a color for the specified color register:

byt0	***color register***
&0	player-missile 0
&1	player-missile 1
&2	player-missile 2
&3	player-missile 3
&4	playfield 0
&5	playfield 1
&6	playfield 2
&7	playfield 3
&8	playfield 4

byt1=hue (**&0-&15**) see Chapter 4 GRAPICS MODES

byt2=luminance (**&0-*&15***)

H. I/O Statements

BGET# *byt0,bin1,bin2*

The **BGET#** statement gets a block of bytes from a file. It is used to input fixed length records.

byt0=channel number
bin1=address to store record
bin2=size of record

Consider the following declarations:

```
BYTE PAYROLL
STRING(32) NAME
FLOAT PAYRATE,HOURS,TOTAL
STRING(9) SSN
```

The storage for these variables is contiguous since their declarations are contiguous. They take up 1+(32+1)+6+6+6+(9+1)=62 bytes. If a record exists on a file with the exact same structure (perhaps created with **BPUT#**, which see), it may be input with the followng statement:

```
BGET# &2,ADR(PAYROLL),62
```

The **BYTE** called **PAYROLL** might be used for a delete flag. You are limited only by your imagination on how you might use this feature. An entire array might be read in:

```
FLOAT TABLE(99)
BGET# &1,ADR(TABLE(0)),600
```

Just remember, **BYTE**s require one byte each, **BINARY**s two bytes, **FLOAT**s size bytes, and **STRING**s require one plus their maximum length.

After an end-of-file error (136), **DPEEK(856)** reports the number of bytes actually read.

BPUT# *byt0,bin1,bin2*

BPUT# is identical to **BGET#** except it does an output instead of an input.

CLOSE# ***byt0***

The **CLOSE#** statement closes a channel. See **OPEN#**.

DGET# ***byt0,varbin1***

The **DGET#** statement gets two bytes from channel ***byt0*** and stores them in ***varbin1***.

DPUT# ***byt0,bin1***

The **DPUT#** statement takes the two bytes of ***bin1*** and outputs them to channel ***byt0***.

ERASE ***str0***

The **ERASE** statement, like **ERASE** in the CP, erases files. It uses channel 7. Wildcards are permitted, but the device must be specified.

FGET# ***byt0,varflt1***

The **FGET#** statement gets six bytes from channel **byt0** and stores them in **varflt1**.

FPUT# ***byt0,flt1***

The **FPUT#** statement takes the six bytes of ***flt1*** and outputs them to channel ***byt0***

GET# ***byt0,varbyt1***

The **GET#** statement gets a byte from channel ***byt0*** and stores it in ***varbyt1***.

INPUT ***var0,var1,...***

The **INPUT** statement inputs strings, converts them to the proper type, and stores them in each variable. Each string must be seperated by one comma or a carriage RETURN. Leading spaces are ignored.

INPUT# ***byt0,var1,var2,...***

The **INPUT#** statement does the same thing as **INPUT** except that ***byt0*** specifies which channel the **INPUT**ing is to be done from.

LINPUT *var0,var1,...*
LINPUT# *byt0,var1,var2...*

The **LINPUT** and **LINPUT#** statements are identical to **INPUT** and **INPUT#** respectively except for one thing. Only carriage RETURNs are used to delimit strings. Commas will be input just like any other character.

NOTE# *byt0,varbin1,varbyt2*

The **NOTE#** statement notes the current location in a disk file. ***Byt0*** is the channel the file is **OPEN** on, ***varbin1*** will receive the current sector number, and ***varbyt2*** will receive the current byte number in that sector.

OPEN# *byt0,byt1,byt2,str3*

The **OPEN#** statement opens a file for input and/or output. ***Byt0*** specifies which channel is to be used. Channel 0 is open to the screen editor (**E:**) by default. Graphics statements use channel 6. Channel 7 is used by the CP and some statements (**ERASE**,etc). Channels 1 to 5 are fully available to the user. ***Byt1*** and ***byt2*** are the auxilary bytes.

Their use varies from device to device, but the following are used for the screen editor (**E:**), the keyboard without echo (**K:**), printers (**P*n*:**), cassette (**C:**), and disk drives (**D*n*:**):

aux1	
4	input
8	output
9	append (disk)
12	input and output (screen editor)
	update (disk)
13	forced read (screen editor)

aux2 is usually 0 for most devices.

Forced read on the screen editor means that an input from **E:** will read the logical line that the cursor is on without waiting for the user to press RETURN. ***Str3*** is the filename. The usual formats are "**C:**", "**E:**", "**S:**", "**Pn:**", "**D*n*:*filename.ext***", "**R*n*:**". The default device is not automatically used. You must explicitly use the **F$DEF** function in the CP.

POINT# *byt0,bin1,byt2*

The **POINT#** statement moves the internal pointer of a disk file. It is used in conjunction with the **NOTE#** statement. ***Byt0*** is the channel, ***bin1*** is the sector, and ***byt2*** is the byte number in the sector.

PRINT *str0*[;]

The **PRINT** statement outputs a string expression to channel **0**. If no semi-colon is appended, a carriage RETURN is also output after the string is. Unlike ATARI BASIC, multiple string expressions seperated by commas are not supported.

PRINT# *byt0,str1*[;]

The **PRINT#** statement acts just like **PRINT** except you also specify which channel to output to (***byt0***).

PROTECT *str0*

The **PROTECT** statement does the same thing the **PROTECT** command in CP does. It protects a file from being deleted, changed, or erased. This statement uses channel **7**. Wildcards are permitted, but the device name must be specified.

PUT# *byt0,byt1*

The **PUT#** statement outputs a single byte (***byt1***) to channel ***byt0***.

RENAME *str0*

UNPROTECT *str0*

The **RENAME** and **UNPROTECT** statements do the same things that their counter parts do in the CP. They use channel **7**. Wildcards are permited, but the device name must be specified.

I. Other Statements

CHAIN *str0*

This statement causes the compiler to start compiling from the specified file name (***str0***). It is for compiling programs whose text is to large to fit into memory at once. The BASIC compiler is one such program. It should be used only at the end of files as it is not the same as the INCLUDE of MAC/65.

DPOKE *bin0,bin1*

The **DPOKE** statement deposits the two byte binary value ***bin1*** at address ***bin0***. The lower order byte is put at ***bin0*** and the upper order byte is put at ***bin0*+1**.

FPOKE *bin0,flt1*

The **FPOKE** statement deposits the six byte floating point value ***flt1*** at address ***bin0***

[LET] *var0=expression1*

The **LET** statement assigns ***expression1*** to ***var0***. ***Expression1*** should be of the same type as ***var0***, but conversion will be coerced on ***expression1*** if it is not. The keyword ***LET*** is optional and is usually omitted.

MOVE *bin0,bin1,bin2*

The **MOVE** statement moves a block of ***bin2*** bytes from address ***bin0*** to address ***bin1*** (**MOVE *from,to,size***). It starts with ***bin0***, then ***bin0*+1**, ... then ***bin0*+*bin2*-1**. Some examples of its use would be to move the character set to RAM for modifying, moving bit images into player-missile storage, or "page flipping" the graphics screen for animation. See also **RMOVE**

POKE *bin0,byt1*

The **POKE** statement puts the byte value ***byt1*** at address ***bin0***.

REM *text*

The **REM** does not do anything. No other statement may come after it on any given line because all the remainder of the line after **REM** is regarded as a **REM**ark. Remarks are for programmers to insert explanations and documentation into a program's text. This makes it easier to modify the program in the future.

RMOVE *bin0,bin1,bin2*

The **RMOVE** statement does the exact same thing as **MOVE**, except it does it in the opposite order: ***bin0*+*bin2*-1** is moved first, then ***bin0*+*bin2*-2**, ***bin0*+*bin2*-3**, ... ***bin0*+2**, ***bin0*+1**, and finally ***bin0***. Both **MOVE** and **RMOVE** are provided so you can handle overlapping between the source and destination blocks without any problem. Consider the following segment:

```
BYTE A(999)
POKE ADR(A(0)),&0
```

```
MOVE ADR(A(0)),ADR(A(0))+1,999
```

It will rapidly clear the array with zeros. If instead **ADR(A(0))** represented a player (also called sprite) and you were trying to move it down one pixel, the above would erase it instead. **RMOVE** would do the desired operation.

SPOKE ***bin0,str1***

The **SPOKE** statement deposits a string value (***str1***) at address ***bin0***.

J. Functions

ABS(*byt*)-->*byt*
ABS(*bin*)-->*bin*
ABS(*flt*)-->*flt*
ABS(*str*)-->*flt*

The **ABS** function computes the absolute value (positive value) of a number. The function considers **BYTE**s over **127** and **BINARY**s over **32767** negative. Hence **ABS(-&1)=ABS(&255)=&1**. If **ABS** is given a **STRING** argument, it converts the **STRING** to **FLOAT** first.

ADR(*var*)-->*bin*

The **ADR** function returns the address of the argument variable. Note, to get the address of an entire array, you must specify the first element of that array:

```
ADR(A(0))
ADR(B(0,0))
ADR(C(0,0,0))
```

ASC(*str*)-->*byt*

The **ASC** function returns the ASCII collating value of the first character of the **STRING** argument. Eg., **ASC("AB")** equals **&65**.

BIN(*any*)-->*bin*

The **BIN** function converts any type to **BINARY**. **BIN(*bin*)** will have no effect and is perfectly permissible. Note, it will not convert negative ***flt*** and ***str*** values.

BOUND(*var0*,*byt1*)-->*bin2*

The **BOUND** function returns the upper bound of dimension ***byt1*** of variable ***var0***. This is useful in parameter passing where the parameter is declared with an asterisk as a bound. For example, players (sprites) might be implemented as an array of either **128** or **256** **BYTE**s, depending on the resolution: you would use this function to determine whether the correct size has been passed for the current resolution. ***Byt1*** can have the value one, two, or three.

BYT(*any*)-->*byt*

The **BYT** function converts any type to **BYTE**. Note, it will not convert negative ***bin***,***flt***, or ***str***. To convert a negative ***bin***, **AND** it with **255** first.

CEXP(*flt0*)-->*flt1*

The **CEXP** function returns **10.0** raised to the **flt0** power.

CHR[$](*byt*)-->*str*

The **CHR$** function is the inverse of **ASC**. It takes a **BYTE** and converts it into the corresponding ASCII (ATASCII) character. The dollar sign is optional.

CLOG(*flt*)-->*flt*

The **CLOG** function computes the base ten logarithm.

DESC(*var*)-->*bin*
DESC(*subprogram*)-->*bin*

The **DESC** function returns the address of the ten byte descriptor of   variable or subprogram. If ***var*** is an array, do not specify any subscripts.

DPEEK(*bin*)-->*bin*

The **DPEEK** function returns the two byte **BINARY** value at the address specified in the argument. It is the complement to **DPOKE**.

ERR

The **ERR** function returns the last error condition raised. It keeps its value until the statement **ERROR &1** is done.

EXP(*flt0*)-->*flt1*

The **EXP** function returns e (**2.718281828**) raised to the ***flt0*** power.

FLT(*any*)-->*flt*

The **FLT** function converts the argument to **FLOAT** type. It is identical in operation to **VAL**.

FPEEK(*bin*)-->*flt*

The **FPEEK** function returns the six byte floating point number at address bin. It is the complement of **FPOKE**.

INSTR(*str0*,*str1*,*byt2*)-->*byt3*

The **INSTR** function finds the position of ***str1*** in ***str0***, beginning at the ***byt2***th character of ***str0***, and returns which character position it starts with. If ***str1*** is not found, ***byt3***, the value returned, is **&0**.

INT(*flt*)-->*flt*

The **INT** function returns the integer portion of the floating point argument. That is, it rounds towards zero.

LEN(*str*)-->*byt*

The **LEN** function returns the length (in bytes) of the string argument.

LOG(*flt*)-->*flt*

The **LOG** function returns the base e logarithm of the argument.

MID[$](*str0*,*byt1*,*byt2*)-->*str3*

The **MID$** function extracts a portion of ***str0***. That portion begins at the ***byt1***th character and has ***byt2*** characters. Exceding the length of ***str0*** is not detected. The dollar sign (**$**) is optional in the function's name.

NOT(*byt*)-->*byt*
NOT(*bin* or *flt* or *str*)-->*bin*

The **NOT** function performs a bitwise inversion of the bits of a **BYTE** or **BINARY** expression. Since true is represented by **&255** and false by **&0**, **NOT(**true**)** is false and **NOT(**false**)** is true.

PEEK(*bin*)-->*byt*

The **PEEK** function returns the **BYTE** value located at address ***bin***.

SGN(*byt0*)-->*byt1*
SGN(*bin0*)-->*bin1*
SGN(*flt0* or *str*)-->*flt1*

The SGN function returns the sign of its argument. That is:

byt0	***byt1***
&128 to &255	&255
&0	&0
&1 to &128	&1
bin0	***bin1***
32678 to 65535	65535
0	0
1 to 32767	1
flt0	***flt1***
<0.0	-1.0
= 0.0	0
>0.0	1.0

SPEEK(*bin*)-->*str*

The **SPEEK** function complements **SPOKE** and returns the **STRING** at address ***bin***.

STR[$](*any*)-->*str*

The **STR$** function converts its argument to a **STRING**. The dollar sign (**$**) is optional.

VAL(*any*)-->*flt*

The **VAL** function is identical to **FLT** and converts the argument to floating point.