Atari Home computer operating system usage guidelines

The document itself begins on the next page.

Document source:

Original backup tapes owned by Dutchman2000, obtained by Atarimania.
Documentary research and PDF layout by Laurent Delsarte.
Note that these backup tapes contain A LOT of information spread out in many folders, meaning it will take time to process the important bits.

Document identification:

Original file name:	MR.SCOTT.00275.DOC extracted from CEO.01JUN84
Title of document:	Atari Home computer operating system usage guidelines
Author(s):	(presumed, to be confirmed) R. Scott SCHEIMAN (Digital circuit design, also Operating systems development)
Original file date:	(unknown) Previous V00213.DOC is dated 1983-11-03 R. Scott SCHEIMAN left Atari probably in late 1983.
Type of document:	Manual
Target audience:	Professional developers
Status:	Very advanced draft
Reference (Atari):	(unknown)
Reference (Laurent Delsarte):	For any discussion, this PDF has been given the reference **BKUP-19XX-XX-XX-MANU-0001A-B** which should be quoted in any communication.
Tags:	#Atari #8bit #6502 #400 #800 #1200XL #600XL #800XL #1400XL #1450XLD #OS #Bug #REV.A #REV.B #REV.10 #REV.11 #REV.1 #REV.2 #NTSC #PAL #BASIC #DOS2 #DOS3

Comments:

As stated in the introduction, this is obviously the draft version of a document which – in this recovered version – has not been finalized:

- Some memory addresses and telephone numbers have not been completed;
- Questions from the author are indicated by ????????;
- Some sections contain only sketches of rough ideas to be developed or, in other cases, no content at all.

Obvious spelling mistakes and typos have been corrected.
Additions by Laurent Delsarte are indicated by [[and]].

Given the impressive number of documents still to be explored, it's not impossible that a more recent version of this same document will resurface in the future.

This page intentionally left blank

Atari Home computer operating system usage guidelines

NOTE: This is only a draft of this manual. It is probably more complete than the final manual will be.

Table of Contents

Disclaimer

Every effort has been made to ensure that this manual accurately documents proper guidelines for use of Atari computers. However, Atari does not guarantee its accuracy. Thus, Atari makes no warranty, express or implied, regarding these guidelines. In particular, Atari makes no warranties of merchantability or fitness for a particular purpose. Atari shall not be liable for any losses or damages of any kind that result from use of these guidelines or any related material. Atari reserves the right to make changes to the guidelines and related materials without notice.

Should you encounter some problem which you find is vague or missing from this manual, you can call Atari customer service at 800-xxx-yyyy. Every effort will be made to provide you with an accurate answer.

Purpose of this manual

This manual is primarily intended to aid the professional programmer who is writing applications software for the Atari home computers. It includes specific idiosyncrasies of the various revisions of the Atari computer, and suggestions and guidelines on how to write programs properly so they will run on all of the machines in the Atari computer line.

The main emphasis is on the operating system (OS). Also included is information on aspects of the hardware which varies from machine to machine, and information about Atari BASIC, DOS 2, DOS 3, and peripherals, all of which are commonly used by applications for the Atari computers.

References to other manuals

This manual is intended to supplement, and be used in conjunction with, the following Atari publications. This manual is written for the person who is already very familiar with what the Atari computer is all about. For the new programmer, however, it still might be best to start here – you may not understand it all at once, but there are useful things here to keep in the back of your mind while studying the other manuals.

- Atari 1200XL Home Computer Operating System Source Code Listing.
  This is a listing of the OS referred to in this manual as Rev. 11.
- Atari 600XL Home Computer Operating System Source Code Listing.
  This is a listing of the OS referred to in this manual as Rev. 2.
- Atari 400/800 Home Computer Operating System Source Code Listing.
  This is a listing of the OS referred to in this manual as Rev. B.
- Atari Personal Computer System Operating System User's Manual, along with the supplement Atari 1200XL Operating System Manual.
- Atari 1200XL Home Computer System Hardware Manual.

Overview/summary

This manual gives specific information about the Atari hardware and operating system environment for applications programs. Following these guidelines as closely as possible will help ensure that you will write programs which will work on all computers in the Atari home computer line, even future versions.

Should you run into a question not covered by this manual, you can probably determine your own answer through common sense. But the basic rule of thumb should be:

DON'T USE ESOTERIC TRICKS!!

The Atari operating system was designed, as are most good operating systems, to be called through clean, defined interfaces. These interfaces (entry points, procedural behaviors, RAM use, etc.) are considered by Atari system designers to be fundamental to allowing applications programmers to write programs which will work on all Atari computers. To our best ability, we have tried to keep all of these interfaces as unchanged as possible as we have upgraded our computers. However, it depends on you, the application writer, to make this work, by using only these defined interfaces. This manual is intended to document and explain most of the important aspects of how an application program should work with the other parts of the system.

OS history

Here is a list of the operating systems Atari has sold in its home computers. They are listed in order of creation, along with known bugs or other behaviors which could matter to you. Each OS was created using the previous one as a base: thus, when a bug was fixed or an enhancement added in a particular OS, then all of the subsequent OS revisions will have that fix/enhancement.

In order to write code to work on all Atari OS's, if your code needs to interact with a part of the OS which has changed because of bug fixes or enhancements, then you should either assume the least common denominator (for example, assume the bug always exists and work around it), or you may "special case" your code (test which OS the code is running with). Techniques for both of these approaches follow.

Rev. A: This is the original OS, used in the 400 and 800 computers. Its major claim to fame is a bug which causes printers to print all or part of a line a second time (intermittently), and disk drives to "go to sleep" for thirty seconds. (Note that you can solve these problems by purchasing a Rev. B OS for your 800. Rev. B can be installed by taking your computer to your computer shop or shipping it to Atari??????????)

Rev. A also contains the following bugs:

When the cassette is opened for output, the header may be written incorrectly. This is because the POKEY hardware reset (after you have hit the SYSTEM RESET key) may leave POKEY in an incorrect state. The manual bypass is to perform some non-cassette I/O before writing to the cassette, such as opening the printer. It will work to try to do I/O to a printer even if you don't have one – just ignore the resulting error message.

When the cassette is opened for output, the header is written immediately and continues to be written until the first data record (128 bytes of data) is written. There is a limited amount of time allowed (about 30 seconds) for the first record, or the tape may not be readable. This has not been fixed in any OS release. Applications programs should have at least 128 bytes of data available before opening cassette for output, and then should write this data immediately after the open.

A spurious character in the OS source, not flagged by our assembler, caused an instruction to be omitted in the interrupt handler. POKEY timer 4 interrupts (VTIMR4) are unusable unless the application supplies its own interrupt handler. The bad instruction is shown commented-out on some published listings of this OS. The work-around is to avoid timer 4 or to trap immediate IRQ and handle the interrupt yourself.

No interrupt vector was supplied for the BREAK key interrupt.

The keyclick routine used the wait-for-sync operation to produce the click. Display list interrupts (DLI's) from ANTIC are delayed by one line by wait-for-sync's, causing, for example, horizontal color changes on the TV to jump one line when the keyboard is used.

The SETVBL subroutine (called through vector SETVBV) disabled all NMI interrupts but when finished, the vertical blank (VBLANK) interrupt was left on and DLI's off. The major user of SETVBV is SIO. Thus, for example, horizontal color changes on the screen "flash" on and off during I/O.

If SETVBV is called to set a VBLANK timer, and a VBLANK interrupt occurs at just that time, the timer could be counted down by two by the time SETVBL returns! To avoid this, don't call SETVBV to set a timer when a VBLANK is about to occur (see under Rev. B).

The BRK instruction interrupt handling was not properly implemented. In particular, if a BRK instruction is executed at the same time a higher priority IRQ occurs, the higher priority interrupt is serviced and the BRK lost. However, the IRQ will probably not return to the proper spot in the code where the BRK was. More recently, a bug has been found in the 6502 microprocessor itself! If, during the processing of the BRK, an NMI (VBLANK or DLI) occurs, the stack is set as if the BRK were processed, but the processor takes the NMI vector! The BRK instruction is only useful if you are writing a debugger. In order to avoid confusion, these problems have not been fixed in any OS. Debugger writers can avoid the first problem by capturing the immediate IRQ vector and testing for the BRK instruction FIRST, then allowing the OS to handle the other IRQ's. Note that the B flag in the P register cannot be trusted when handling IRQ's (another 6502 quirk!); instead, you must look at the B-bit in the stacked copy of the P-register. To avoid the concurrent BRK/NMI problem, you must add code to NMI handling to check for the B-bit. Better yet, avoid using the BRK altogether!

Bugs exist in the OS disk handler. Atari disk operating systems have the proper bypasses – the low-level disk handler should not be used by applications programmers.

A bug existed in SIO where, if it was told to use an I/O buffer which ended on a page boundary, SIO went into an infinite loop. A patch for this has been placed in the Atari DOS 2 and DOS 3 products. When not using the DOS, avoid buffers which end on a page boundary.

Certain screen-control registers are shadowed in the OS database and written to ANTIC and the CTIA/GTIA during each VBLANK period. This was done during a period where IRQ's were allowed. If an IRQ occurred which required too much time to process, the VBLANK shadowing could occur during the visible part of the following TV frame, possibly with visible results (the screen flashed occasionally). The fix is to avoid long IRQ routines (a good practice in general!).

The screen handler sets up the display area and display list near the top of RAM. Often, there is an unused area just above this are a and below the actual top of RAM. However, when the screen is re-opened (which happens, for example, when SYSTEM RESET is pressed), the screen handler zeros out not only its display area but some of the area above. Therefore, it is not a good idea to put data in this area unless you don't mind its being cleared. This problem was fixed in a later revision.

If, in a non-scrolling character screen mode (in Atari BASIC, this means a screen mode other than mode 0), a character was just placed in the last position on the screen and a clear screen character is output, rather than clearing the screen, an error 141 occurs. This is fixed in a later revision.

Two small problems with CIO error handling exist; they are fixed in a later OS revision. First, when CIO is asked to read an input record which turns out to be larger than the supplied buffer, no EOL appears in the data. The appropriate error code is returned, but code which ignores the error and simply looks for the EOL will fail. Secondly, when using byte-at-a-time I/O (data in the A-register), the zero in the length field which signifies this A-register transfer is set to a nonzero value by CIO (but only when an error occurred during the I/O). Thus, it is best to reset the zero in the length field before calling CIO each time.

Rev. A PAL: PAL is the name of the European TV standard, and refers here to special versions of the Atari OS for European computers. Rev. A PAL is the same as Rev. A except that some timing constants were changed in cassette handling and keyboard repeat rate to account for the 50 Hz European VBLANK rate (60 Hz in America [[USA, NTSC]]).

Rev. B: Fixed the printer multiple print and disk sleeping bugs.

Added a vector for the BREAK key (xxxxx). Protected the appropriate shadowing code in the VBLANK section by not re-enabling interrupts too soon. Fixed the POKEY timer 4 problem. Fixed the SIO problem with buffers ending on page boundaries. If you are writing code to work on Rev. A, none of these fixes should make any difference to you.

The SETVBV (SETVBL) routine was re-coded to avoid the bad interaction with VBLANK. Specifically, it checks to see if a VBLANK is near. If a VBLANK is near, SETVBL loops until the VBLANK has passed. This technique should be copied into your code if you intend to use SETVBV with Rev. A OS's and the old bug could be a problem for you. Here is the code:

```
STA $xxxx   ;This synchronizes just before the interrupt,
            ;  if it is going to occur...
LDX #5
LOOP: DEX   ;And this loop continues until just after
BNE LOOP    ;  the VBLANK interrupt (so what follows
            ;  WON'T be interrupted)...
```

You may precede any call to SETVBV (which you feel would be affected by the Rev. A bug) with the above code. There will be no problem if you do this on Rev. B or the other revisions (except for the added time to process the SETVBV).

NOTE: The above technique can be fouled if IRQ interrupts are happening. If this is critical to you, ALWAYS disable IRQ's (SEI) before calling SETVBV (before the above code if you are using it), and then re-enable the IRQ's (CLI) just after the SETVBV call. SETVBV does not include the SEI and CLI in any OS revision.

In making these fixes, some code was moved in the OS ROM's. Programs which made jumps directly into OS code (a practice which Atari considers an illegal use of the OS) using the Rev. A OS fail with Rev. B. In fact, there are one or two programs in existence which, through luck, made such "illegal" jumps but still appear to work – but they are not working the same way!!!

(Rev. B PAL: This is to Rev. B as the Rev. A PAL is to Rev. A. However, no copies of this OS revision have been produced. Information is included here in case this OS comes to market.)

Rev. 10: This is the OS first released on the Atari 1200XL. Its major problem is incompatibility with a lot of non-Atari applications which violated good programming practice (and some Atari programs, too). Included in this list of incompatible programs were 40 or so cassette titles sold through the Atari Program Exchange (APX) whose auto-boot header program made improper assumptions about the OS environment which Rev. 10 did not support. This problem has been addressed by releasing the Rev. 11 OS, and by re-mastering the cassettes using an auto-boot program written to follow the proper rules.

Nearly all of the code in the OS moved between Rev. B and Rev. 10. Many changes were also made to the RAM database, necessitated by Atari's development methods. Every effort was made to leave all SUPPORTED RAM and ROM interfaces unchanged. However, it has turned out that a significant number of non-Atari programs use UNSUPPORTED features of the older OS and therefore do not work.

The following changes were made for the Rev. 10 OS:

Game controller ports 3 and 4 were removed from the computer starting with Rev. 10. The RAM shadows for joysticks and pots (PIA port B and POKEY paddles 5 through 8) are now shadowed from PIA port A and paddles 1 through 4. In other words, if you are writing a game for all 4 controller ports, go ahead, just as long as you use the RAM shadows rather than reading the hardware registers directly. This applies only if the players take turns – since port 1 is mapped into the shadows for port 3, and port 2 is mapped into port 4. On a 400 or 800 computer, players would take turns normally. If your game is being played on a 1200XL or later machine, when player 3 gets his turn, he just uses the controller plugged into port 1, and so on.

The second game cartridge slot is not looked for by the Rev. 10 OS.

The OS was modified to automatically switch between American [[USA, NTSC]] and PAL timings, based on the machine configuration bit in the GTIA. There are no special PAL versions of the OS beyond Rev. 10.

The SYSTEM RESET key is supported by Rev. 10 as being connected either to the ANTIC reset input or to the 6502 reset. (The 1200XL and subsequent computers attach SYSTEM RESET to the 6502.) In order to maintain the old function of SYSTEM RESET, called "warm" start, it is necessary to distinguish a press of this key from a fresh power-up (called "cold" start). This is done by storing certain constants in three RAM bytes. On fresh power-up, these constants are not found in the RAM by the OS, which then does a cold start. If the constants are found, a warm start is performed. Changing any of these bytes (xxxxx,yyyyy,zzzzz) from its special value will "prime" the system to force a cold start on the next press of SYSTEM RESET. (A better and more general way to force cold start on SYSTEM RESET is to set the OS variable COLDST to xxxxx.)

An OS identification field was added to the OS ROM's. More on this later.

Support for the five new keys (F1 through F4, and HELP) was added, as was the ability for an application program to supply its own keyboard translation table to redefine the meaning of the keyboard.

The international version of the ATASCII display character set was added, which replaced the 29 graphics characters with 29 international letters and punctuation. Note that this change is invisible to any application (except next paragraph) – the character set remains the same (typed the same way, handled by BASIC the same, and so on); only the way it is displayed to the screen is changed. In addition to this OS change Atari has introduced printers capable of printing the 29 "graphics" characters using this new international font.

One of the functions added to support the new function keys was the ability to switch display context between the old ATASCII and new international ATASCII display modes. This is handled by one of the function keys, simply by changing the RAM shadow CHBAS (xxxxx) to the appropriate character set. Unfortunately, this key only chooses between the two OS character sets, and applications programs which define their own character set are messed up should the user hit this function (there is no good reason for him to do so, of course). A good fix for this is to reset CHBAS during every VBLANK rather than just once during initialization.

A number of new features were added which do not directly affect applications programs (but might affect your user's instruction manuals). Refer to the 1200XL OS User's Manual Supplement for details.

The printer handler was modified to support device numbers (i.e., P2: is distinct from P3:). A number of Atari printers support this unique addressing, allowing more than one type of printer to be simultaneously attached. Unfortunately, the fact that Atari BASIC bypasses CIO for PUT's was inadvertently overlooked. This bug was not found for some time, and therefore not fixed until Rev. 2. The effect is that, if I/O is done to some non-printer device (such as a disk), then a subsequent PRINT or PUT to a printer "inherits" the device number from the previous disk access, rather than using the device number supplied at the time the printer was OPEN'ed through the IOCB in use. The simplest fix is to CLOSE and re-OPEN the printer each time a PRINT is needed. If this messes up the print formatting (i.e., you are using the comma or semicolon at the end of your PRINT statements) the fix is to POKE the proper device number to location xxxxx just prior to doing a PRINT. Both of these fixes will work on all the OS's.

The printer handler was modified to print the final buffer contents when the IOCB is closed (unless the buffer is empty). Thus, if you BREAK out of a program and then re-start it, the IOCB closing and re-opening at the start of the program no longer results in the not-yet-printed data from the previous run printing first on your new run.

CIO error handling of input records truncated because the input buffer is too short for an entire record was modified to place an EOL as the last character of the buffer. Also, the IOCB length field is left zero when it was set zero by the CIO caller and an error occurs.

When a clear-screen character is written to the screen, the screen is cleared no matter where the cursor is placed.

One small change was made to the floating-point package: now an error occurs when one attempts to take the LOG or LOG10 of zero.

A screen editor bug was uncovered in Rev. 10: while doing continuous output to the screen (as, for example, when listing a BASIC program) the BREAK key is occasionally "missed." Actually, the character being written to the screen is NOT displayed and the BREAK key is not reported to the screen editor's caller. Thus, when the user hits BREAK again, a character is missing from the screen. This bug was "uncovered" rather than "introduced" by Rev. 10 because its cause actually existed in Rev. A! The problem turned out to be timing dependent, and interrupt handling changes in Rev. 10 caused the first appearance of this problem.

Rev. 11: Like Rev. 10, this OS is for the Atari 1200XL computer. It was released almost immediately after Rev. 10. Its major purpose was to provide bypasses for the problems with the bad APX cassette tape programs. Additionally, some other small problems were corrected. Unfortunately, the short production life of the 1200XL has caused far more Rev. 10's than Rev. 11's to have been shipped. Rev. 11 is available from Atari on a factory retrofit basis???????

It was found that the circuitry for the SYSTEM RESET key had electronic or mechanical bounce problems far beyond what OS Rev. 11 was designed to handle. The OS reset operations are not re-entrant. In particular, the operation of determining the size of RAM memory changes RAM cells, tests them, then changes them to their original values. A reset bounce before a RAM cell is changed back leaves it permanently modified. This problem also shows itself by intermittently causing a cold rather than a warm reset. The only known fix [[is]] to retrofit Rev. 11 into your computer, which solves the problem by starting the reset operation with a 1/10 second delay to allow the bouncing to stop before non-reentrant work is done.

The Rev. 10 problem with swapping characters sets out from under unsuspecting programs when the user types the wrong function key was "solved." Rather than forcing CHBAS, a new database variable (CHSALT, $026B) is swapped with CHBAS each time the function key is used. Thus, if the user accidentally switches to the alternate character set, he can press the function key to restore the proper character display.

Small fixes were added to features which were new as of Rev. 10. The new smooth scrolling screen editor feature had a problem which caused the screen to flash on and off during SIO activity, and the OPEN/CLOSE logic did not handle the display list quite correctly. Rev. 11 fixes these problems.

Very little code in the OS stayed in place. These few "small" changes required that many of the OS modules be scattered in pieces in order to fit the requirements of a quilt work "jigsaw puzzle." A rev. 11 listing will look very much like a Rev. 10 listing, except the code addresses will be different.

Rev. 1: The Rev. 1 OS represents a relatively minor modification to the Rev. 11 OS. Rev. 1 is the original operating system written for Atari's newest line of computers: the 600XL, 800XL, 1400XL, and the 1450XLD. Rev. 1 has been superseded by Rev. 2, and in a very short time Rev. 2 should vastly outnumber Rev. 1. Note that the same OS is used in this entire new line of computers.

The Rev. 1 OS is simply the Rev. 11 OS changed to support the new features of the machine: the parallel I/O bus, and the built-in BASIC cartridge. From the point of view of the applications programmer, these features should not make this OS appear different from the previous ones.

The parallel bus is supported by the OS as if I/O devices were plugged into it, just as the OS supports I/O devices through the serial bus. From the applications point of view, calls to CIO (PRINT, INPUT, PUT, etc. in BASIC) work just as they always have.

Similarly, the built-in Atari BASIC appears to an applications program just as if it were the BASIC cartridge. However, when designing your program, pay attention to how the built-in BASIC is disabled if you don't want it: the computer user is instructed to hold down the OPTION key while turning on the computer. You might want to include a note about this in the instruction book you write for your program. Also, be aware that he will be using the OPTION key when your program starts. You should consider ignoring the OPTION key at the beginning of your program.

The screen-editor/BREAK key interaction problem is fixed in the Rev. 1 OS.

Rev. 2: Rev. 2 followed Rev. 1 for the newest XL line of computers almost immediately. It fixes the Rev. 10 problem with the printer handler "inheriting" the device number from previous I/O when called from Atari BASIC.

How to Tell Which OS Your Program is Running With: If you need to special-case your code to work with or around a feature or bug which exists only on certain versions of the system, you may use the following tests:

Rev. A, Rev. A PAL, Rev. B, and (possibly) Rev. B PAL can each be uniquely identified by looking at the checksum field in the OS ROM. In order to ensure a valid test, you must look at BOTH bytes of the checksum. Atari will ensure that any two-byte checksum will be [[NOT]] re-used in a future OS revision; however, any single byte of any checksum may reappear. Checksums for Rev. B PAL are included here for reference, but are NOT guaranteed. The Rev. B PAL currently ready for release may NOT match the Rev. B PAL actually released in the future. This is unlikely, however.

Checksums for the newer revisions are included for completeness; however, there are other techniques recommended to test for the newer versions.

	REVISION CHECKSUM ($FFF8) & ($FFF9)	
A	$DD	$57
A PAL	$D6	$57
B	$F3	$E6
(B PAL	$22	$58)
10	$BF	$E5
11	$40	$77
1	$18	$6B
2	$8C	$6C

There is a very simple test to tell the difference between the first four operating systems and all subsequent revisions. In revisions A, A PAL, B, (and B PAL), the location $FCD8 will have the value $A2. It will not have that value in any subsequent OS. Since the techniques for checking what OS you have differ between these first four OS's and all subsequent ones, this is a good test to make first to see what tests are valid next.

Starting with the Rev. 10 OS, an OS identification area appears in all OS's. For the OS's released to date, this field has the following interpretations and contents:

ADDR	REV 10	REV 11	REV 1	REV 2	MEANING
$FFEE	$26	$23	$11	$10	BCD release day
$FFEF	$10	$12	$03	$05	BCD rel. month
$FFF0	$82	$82	$83	$83	BCD rel. year
$FFF1	$01	$01	$02	$02	CPU series
$FFF2	'AA'	'AA'	'BB'	'BB'	OS version
$FFF4	0,0,0	0,0,1 (see text)	0,0,0	0,0,1	'name'
$FFF7	10	11	1	2	Atari 'internal' release number
$FFF8 (two-byte checksum, see above)					

NOTE: in the above table, $ indicates a hexadecimal value, quotations denote 7-bit ASCII with the most significant bit zero, and numbers without $ are decimal. BCD means binary coded decimal.

If you know that some change occurred at a particular version, and exists in all subsequent versions, the release date field will always serialize the Atari OS's. The CPU series (a field which originally had a different meaning) is 1 for the 1200XL and 2 for the 600XL, 800XL, 1400XL and 1450XL (since all these computers use the same OS they must have the same series number). The interpretation of this field in future OS revisions is NOT guaranteed, however. Likewise, the apparent pattern in the version 'name' field does not guarantee Atari's use for this field in the future. You can see here where the names given to the various OS's in this booklet come from: internal release numbers are assigned during the OS testing and debugging cycle. Each OS maintains the final cycle number (called internal release number above) when it is released for production.

Atari BASIC

More than half of the applications for the Atari home computers are written in BASIC. Therefore, it is as important to know the ins and outs of using BASIC properly as it is to know the OS rules.

Like the OS, BASIC has been released in more than one version. Rev. A BASIC, the first one released, is probably most known for a bug where, when editing a program, the program gets "messed up". Additionally, Rev. A BASIC has two other bugs. First, when moving any character string (or substring) whose length is a multiple of 256 bytes, the move is performed incorrectly. ("Moving" a string, probably better called "copying", is what a LET statement does.) To avoid this problem, either use strings which never reach 256 bytes in length, or carefully code your program to move strings in terms of substrings which are less than 256 bytes (or at least not multiples of 256). The other bug involves the PRINT statement: If a PRINT statement ends with a character-string having as the last character either the graphics character CTRL-R or the graphic CTRL-U, then BASIC behaves as if the PRINT statement also had a semicolon at the end of the PRINT statement. This only occurs if the CTRL-U or CTRL-R is in a character-string constant (not a character variable). Since this problem is fixed in Rev. B, adding an extra PRINT to provide the effect of an EOL will cause a blank line to print if your program is run under Rev. B BASIC. A better solution is to make sure no print ends with CTRL-U or CTRL-R (for example, print a blank space after the CTRL-U or CTRL-R).

Most of the other problems are numerical precision problems – very normal for floating-point number representation, but very disconcerting to people who do not understand the behavior of such numbers.

Rev. B BASIC is now available in cartridge form, and Rev. B is also the version being used in all the new CPU's which have BASIC built in. This BASIC fixes the string move and editing problem. Some of the other fixes are: Integer numbers raised to positive integer powers give accurate results to 10 places; LOG and CLOG of 1 are now zero; numbers now print with no more than 9 decimal places; SQR of.001 now gives a positive number. The other fixes involve syntax checking – Rev. B gives error messages in those cases where Rev. A would lock up.

Rev. B BASIC, unfortunately, introduces a new problem, and, in fact, Rev. C is on its way. Rev. B requires a small amount of extra internal data, which reduces the area available to the application program. While this area is very small (just 16 bytes), there have already been discovered a few applications programs which will not load or run because they were within 16 bytes of using all of memory (with the older BASIC). Additionally, there is a problem with this 16-byte area: during the development of a program, this area gets saved whenever a program is SAVEd. When re-loaded, this saved version gets re-loaded as part of the program area (not the active 16-byte area at re-load time). In other words, these inactive 16-byte areas "pile up", one for each time the program is saved and re-loaded, thus taking away user program space. The solution to this problem is to use LIST and ENTER whenever you want to get rid of these dead areas.

Fortunately, it is very easy to write BASIC programs which will work with all the various versions of the system. Assuming no programming "tricks" are used (read on), you should first write your programs as described above to work with both BASIC revisions (string move and PRINT problems). Then, just in case, allow a small amount of room for possible changes in the BASIC cartridge, or DOS. A few dozen bytes should suffice for this. Additionally, if you expect to sell your program through APX, leave xxxxx bytes for the copy protection code added when your program is prepared for production.

If your program will use NOTE and POINT, read the section in this booklet about DOS 2 and DOS 3.

If your program will use PEEK or POKE statements, or if it will load or POKE an assembly language subroutine, then you essentially have to follow all the rest of the guidelines suggested in this booklet. You are essentially doing what you could do in assembly language when you use PEEK and POKE. Also, pay particular attention to the memory use rules when deciding where to put your assembly subroutine.

About the hardware

Only the Atari 800 has a second game cartridge slot (commonly referred to as "Slot B"). OS revisions 10 and beyond do not even make a check for the presence of the "B" cartridge.

The hardware senses the presence of a cartridge and automatically disables RAM under the cartridge (in 8 K-byte chunks). However, there is no simple signal available to a program to determine whether a cartridge is present or not. To test for a cartridge (if, for example, you want to give an error message if your program requires that there is no cartridge) use the following technique, which is what the OS does. First, check RAMSIZ ($02E4). If it is $B0 or greater, then the cartridge is not there (because RAM is there). If there is not RAM, check for a cartridge by reading CART ($BFFC). By convention, all cartridges have the value ZERO ($00) at $BFFC. If there is no cart, reading $BFFC will produce a nonzero result.

It is not a safe practice to make any check for a specific cartridge (by assuming you will find some particular value at some particular location) because application cartridges might be revised, just as the OS has been revised. This is even more likely for programming language cartridges, such as BASIC. Similarly, just as with the OS, it is unsafe to assume that subroutines you find inside a cartridge will always work if you call them yourself, or that the location or meaning of variables which are "internal" to the cartridge will never change, etc.

There are some "feature differences" among the machines. Among these are the 16K memory limit of the Atari 400 (so you should strongly consider ensuring that your program works in this space). The HELP key and the programmable function keys appear only on some Atari models (so you should program "aliases" if you use these keys). The two general-purpose LEDs only appear on the 1200XL (more detail below). Game ports 3 and 4 only appear on the 400 and 800 models (more below). The parallel I/O bus appears only on the newer machines, and the ability to add hardware in the "RAM slots" is only available on the 800. There is a proper protocol for using the parallel I/O bus – this bus is discussed in another Atari document.

You are encouraged to make use of machine features such as the HELP and programmable function keys. This will make your software more "appealing." It is best, however, to provide alternate keys for your users who do not own a version of the computer which has the function you have chosen to use. For example, if you program provides help, use the HELP key, but also allow another key to do the same thing (e.g., ESC or the reverse-video key). In this way, you do not limit the market for your application.

More subtle differences may exist among the machines. For example, various keyboards of different manufacture have different "bounce" characteristics. De-bouncing is handled for the main keyboard by the operating system (another good reason not to bypass the OS). Be sure you do your own good job of de-bouncing the START, SELECT, and OPTION keys. Another difference is that the peripheral bus lines are electrically different among the machines. For example, not all of the machines have the +12 volt line, and the AUDIO line has different characteristics on different machines. None of these differences should affect any of your software, however.

Keyboard de-bounce was just mentioned as a good reason to use the OS keyboard-handling code (K: or E:). Another good reason is that the OS allows machine enhancements to appear to work for all applications, without having to revise the applications programs to keep up with machine changes. The best example of this so far is the addition of the four new function keys. These keys' default function is to act just like the four cursor controls (up, down, left, right). This default function only works for applications which read the keyboard through the OS – programs (quite a few exist) which read the keyboard directly through POKEY are unaware of the four new keys, so this "default" does not work in these cases.

Speaking of the keyboard, there is something more to watch out for. Never write a program which uses both CTRL and SHIFT together with another keyboard key. You may already know that this combination works with some keys and does not work with others. This comes from the way the keyboard is wired. Which keys "work" with CTRL and SHIFT together, and which ones don't "work", is NOT guaranteed by Atari. So, you may find that a program you write which uses such key combinations won't work on some future Atari computer because the CTRL-SHIFT combination you chose gets locked out.

Game controller ports 3 and 4 do not appear on the 1200XL or any subsequent machine (corresponding to OS Revisions 10 and beyond). There are two ramifications of this: the functional implications of not being able to plug devices or controllers into ports 3 and 4, and the re-definition of both the PORTB of the PIA and of paddles 5 through 8 for POKEY.

For non-game applications, it will probably turn out that if you want your application to work on the newer machines you should forget about ports 3 and 4. Some games, however, can be written to work on all the machines, without your software having to know or care whether ports 3 and 4 exist and are actually being used, or whether ports 1 and 2 are "standing in" for 3 and 4. Specifically, this will work if the game involves taking turns; it won't work if a player on port 3 or 4 needs to interact with the computer at the same time players on port 1 or 2 are interacting. All revisions of the OS copy game controller information into the POT, STICK, and TRIG variables in the low RAM OS data area, once every VBLANK time. Starting with Rev. 10, those controller "shadow" variables which correspond to ports 3 and 4 are filled by looking at 1 and 2. So, if your game looks at the shadow for a controller in port 3, player 3 can use your game just fine on a 1200XL if he just uses the controller player 1 was using (the controller attached to port 1). The shadows for ports 1 and 2 work the same as always, of course. By the way, if you take advantage of this "trick", it would help your users to mention in your user's guide how they should "trade off" their controllers if they play your game on one of the newer machines.

POTS 5 through 8 and PIA PORTB (and PBCTL), which used to interact with ports 3 and 4, have been re-assigned in the newer computers. The pots have been assigned a new input function generically referred to by the OS variable JMPERS. This is intended for Atari use only, and has not been used consistently among the machines anyway. If you run across any references to this in Atari manuals, just ignore the whole thing.

PORTB is now permanently assigned as a control port (output only). It is intended to be under the control of the OS only, but you might find your own use for these functions. Follow these guidelines, however. First, never reconfigure this port yourself (it must always be an output register). The 8 bits in the register are assigned the following functions:

```
PORTB BIT  WHEN "0"  WHEN "1"

        7  Self-test ROM enable/disable (Note 1)
        6  Reserved
        5  Reserved
        4  Reserved
        3  LED 2 ON  LED 2 OFF (Note 2)
        2  LED 1 ON  LED 2 OFF (Note 2)
        1  BASIC ON  BASIC OFF (Note 3)
        0  OS-area RAM ON OS enabled (Note 4)
```

NOTE 1: Revisions 10 and beyond. This bit should be used by the Atari OS only.

NOTE 2: Revisions 10 and beyond. However, the LED's themselves appear only on the 1200XL; there is no plan to include these LEDs on any future product. The OS assigns meanings to these LED's: LED 1 indicates keyboard disabled, and LED 2 indicates international character set. These functions are controlled by the OS keyboard handler, corresponding to two of the programmable function keys. You may use these LEDs in your application, but there is no good way to disable the OS use of the LEDs. In other words, if you use the LEDs, include a note in your user's guide that the user should not use the keyboard disable or international character set functions.

NOTE 3: Switching built-in BASIC on and off should generally be reserved for the OS. This operation is under user control: holding the OPTION key during the power-up cycle disables the built-in BASIC. You may enable/disable the built-in BASIC yourself, but if you do it via the PORTB register, the OS will re-establish the original state if the user presses RESET. A better way to disable/enable BASIC is to force the value of the OS flag BASICF ($03F8). Set this to 0 to enable the BASIC; set to 1 to disable. Then "fake" a RESET, to allow the OS to establish the appropriate BASIC-or-no-BASIC environment by jumping to the warm-start vector $E474. Note that a real cartridge plugged into the game slot always takes precedence over the built-in BASIC, regardless of PORTB bit 1. There is no way from software to disable a plugged-in cartridge.

NOTE 4: A feature of all machines built to-date, from the 1200XL (Rev. 10 OS) through the 600XL/800XL/1400XL/1450XLD family (Rev. 2), is the ability to "turn off" the OS ROM and enable the RAM underneath. In the 600XL, this only applies if the RAM expansion is connected. This feature is not supported by the OS; it will be used by "substitute" operating systems, or special applications which do not use the OS or include their own OS. The use of this feature is somewhat complex, and is covered in a separate Atari publication.

If you do have a need to modify a bit in PORTB, it is important that the other bits be left alone. Always first read PORTB (e.g., load the A-register from PORTB), then mask the bit you care about ON or OFF and store the result back into PORTB. There is no RAM "shadow" for the PORTB register.

Memory use

Page zero is divided into two equal parts. The lower half of page zero is reserved for the operating system. With the exceptions mentioned below, NONE of this half of page zero may be modified by an applications program. Similarly, certain of the data in this lower half of page zero are for OS internal use only, so the meanings are subject to change if the OS is revised. The second half of page zero is for use by the application – but be careful: if your program is in BASIC, the "application" is the Atari BASIC interpreter. This will be explained in more detail.

Following is a list of "external" OS variables in page zero. These are either status bytes which are of sufficient use to applications that they will always have the same meaning, or communications variables which can be modified to tell the OS something. The actual use of each of these is described in more detail in the OS User's Guide.

NOTE: other variables not listed are also guaranteed, but they are for Atari use only so are not listed here.

Addresses in the first column are in hexadecimal:

```
$02 CASINI:.RES    2           ;CASSETTE INIT LOCATION
$07 CMCMD:.RES     1           ;SPECIAL USER/OS FLAG BYTE
;
;
$08 WARMST:.RES    1           ;WARM START FLAG
$09 BOOT?:.RES     1           ;SUCCESSFUL BOOT FLAG
$0A DOSVEC:.RES    2           ;DISK SOFTWARE START VECTOR
$0C DOSINI:.RES    2           ;DISK SOFTWARE INIT ADDRESS
$0E APPMHI:.RES    2           ;APPLICATIONS MEMORY HI LIMIT
;
;
$10 POKMSK:.RES    1           ;SYSTEM MASK FOR POKEY IRQ ENABLE
$11 BRKKEY:.RES    1           ;BREAK KEY FLAG
$12 RTCLOK:.RES    3           ;REAL TIME CLOCK (IN 16 MSEC UNITS)
;
$20 ZIOCB   =*                 ;ZERO PAGE I/O CONTROL BLOCK
$20 ICHIDZ:.RES    1           ;HANDLER INDEX NUMBER (FF = IOCB
                               ;FREE)
$21 ICDNOZ:.RES    1           ;DEVICE NUMBER (DRIVE NUMBER)
$22 ICCOMZ:.RES    1           ;COMMAND CODE
$23 ICSTAZ:.RES    1           ;STATUS OF LAST IOCB ACTION
$24 ICBALZ:.RES    1           ;BUFFER ADDRESS LOW BYTE
$25 ICBAHZ:.RES    1
$26 ICPTLZ:.RES    1           ;PUT BYTE ROUTINE ADDRESS - 1
$27 ICPTHZ:.RES    1
$28 ICBLLZ:.RES    1           ;BUFFER LENGTH LOW BYTE
$29 ICBLHZ:.RES    1
$2A ICAX1Z:.RES    1           ;AUXILIARY INFORMATION FIRST BYTE
$2B ICAX2Z:.RES    1
;
$41 SOUNDR:.RES    1           ;NOISY I/O FLAG. (ZERO IS QUIET)
$42 CRITIC:.RES    1           ;DEFINES CRITICAL SECTION
                               ;(CRIT=NON-ZERO)
;
```

```
$4D ATRACT:.RES    1           ;ATRACT FLAG
                               ;COLORS)
;
$52 LMARGN:.RES    1           ;LEFT MARGIN (SET TO 2 AT POWER ON)
$53 RMARGN:.RES    1           ;RIGHT MARGIN (SET TO 39 AT POWER ON)
$54 ROWCRS:.RES    1           ;CURSOR COUNTERS
$55 COLCRS:.RES    2
$58 SAVMSC:.RES    2
$5A OLDROW:.RES    1
$5B OLDCOL:.RES    2
$60 FKDEF:.RES     2           ;POINTER TO FUNCTION KEY DEF TABLE.
$79 KEYDEF:.RES    2           ;POINTER TO KEY DEF TABLE.
```

FKDEF and KEYDEF do not exist before the Rev. 10 OS. They are described in the 1200XL OS Manual Supplement.

CMCMD ($07) is a new use of a location which the OS only uses as a temporary location during initialization. The new use is as a flag byte for communications between certain I/O handlers and calling applications. In general, this byte is now reserved as a general-purpose communications byte. So far, it has been used only by modem I/O handlers. It is unlikely the OS itself will ever interact with this byte. Note that it is not initialized by the OS, and it will also be garbaged by RESET. If more than one caller/subroutine pair will be communicating through CMCMD, you should take care that CMCMD is being used for only one function at a time. It may become necessary to set the value (rather than assume it is still OK) immediately before each subroutine or I/O call where CMCMD is used. CMCMD may be used in this way no matter which OS revision is being used.

The upper half of page zero is reserved for the "user." If your application is written in assembler language and assembled with the Atari Macro Assembler (AMAC), the upper half of page zero is all yours (except if you use the floating point subroutines – see below). If your program is in BASIC (or PASCAL or the Assembler/Editor debugger, etc.) or if you run your program along with another program, the top half of page zero may very likely have to be shared among the various programs. With Atari BASIC, ALL of the top half of page zero is reserved for the BASIC interpreter. Refer to the appropriate technical or user manual for whatever program you are using along with your application.

The floating point package uses bytes $D4 through $FF. These are available for your program if you do not use the package. They are not available if you are using Atari BASIC, since BASIC uses the floating point package.

If you need a few zero page bytes but your program is running with other programs which use all of page zero (e.g., the OS and BASIC) then you may consider "borrowing" some bytes. The best bytes to choose for this are OS variables from the list given above, so you can be sure their meaning hasn't changed if your program is used on some future OS revision. Your program must save the data (the page 1 stack is best for this) before using the variables, and then restore the values for the OS. The tricky part is knowing when the bytes are not being used by the OS or some other program in the system. It is not safe to use any variables the OS may use during interrupt processing. This includes all the screen editor variables, since the screen editor may be invoked any time the user types on the keyboard. Likewise, SYSTEM RESET may be thought of as an interrupt that may happen any time, so the DOSINI and DOSVEC vectors shouldn't be used. Essentially, what is left is the zero-page IOCB (ZIOCB through ICAX2Z). The zero-page IOCB is used only when CIO is called; since CIO is not re-entrant, the zero-page IOCB is free as soon as CIO returns. If you follow these rules, you may "borrow" these zero-page bytes either in main-line code or in interrupt routines. (If you interrupt while CIO is active there is no problem, if you do two things: save and restore whatever you use, and disallow further interrupts – if CIO is active, it is possible that the "true" ZIOCB values may be needed by CIO/SIO interrupt handlers. If your interrupt handler keeps interrupts disabled, the CIO/SIO interrupt will be locked out and therefore won't get confused by finding "your" data in the ZIOCB.)

Page 1 is the 6502 hardware stack, used for subroutine calling and interrupt handling. The stack grows from the high addresses downward, always staying in page 1 (in other words, the stack wraps around from the lowest address back to the top). In theory, the initial value of the stack pointer does not matter (where the "bottom" of the stack is). Another way of saying this is that the stack addresses are circular or form a "ring." Address wrap-around is not the same as stack overflow: stack overflow occurs when more than 256 bytes are pushed onto the stack (assuming you ultimately intend to pop the stack all the way back). In practice, for nearly all applications (including the Atari BASIC interpreter) the stack never grows very big. Furthermore, the initial value of the stack pointer is set by the OS (and the Atari BASIC interpreter) to $FF, so the stack always starts at the high address ($01FF). The lower address of the stack area may therefore be used for temporary variables BUT USE THIS OPTION ONLY AS A LAST RESORT, AND WITH EXTREME CARE. The maximum size to which the stack grows depends not only on the application running in the system, but also on dynamic factors such as user input or I/O interrupt loads (future I/O devices will probably affect this). It will therefore be difficult to know that the stack won't overflow into your variables. Test THOROUGHLY and even then, leave a large slop factor. If you are writing a subroutine for general use, and do not know what applications will use it, all bets are off, since you cannot predict the maximum stack size. Furthermore, just like page 6 (see below), other co-resident programs may already be using this area. Presently, Atari DOS 3 uses this area during initialization, and the Atari TRANSLATOR uses the area continuously. If you are co-residing with a non-Atari program, check it carefully.

Pages 2 through 6 are shared between the OS and application in just the same way page 0 is. Most of these pages is used by the OS, just like the lower half of page zero. Following is a list of the variables in pages 2, 3, 4, 5, and 6 which can be accessed by applications. They are status bytes which are of sufficient use to applications that they will always have the same meaning, or communications variables which can be modified to tell the OS something. General use of these RAM locations is discussed immediately following this list. The actual use of each of these is described in more detail in the OS User's Guide.

NOTE: other variables not listed are also guaranteed, but they are for Atari use only so are not listed here.

```
;       PAGE 2 RAM ASSIGNMENTS
;
$200 VDSLST:.RES    2    ;DISPLAY LIST NMI VECTOR
$202 VPRCED:.RES    2    ;PROCEED LINE IRQ VECTOR
$204 VINTER:.RES    2    ;INTERRUPT LINE IRQ VECTOR
$206 VBREAK:.RES    2    ;SOFTWARE BRK (00) INSTRUCTION IRQ VECTOR
$208 VKEYBD:.RES    2    ;POKEY KEYBOARD IRQ VECTOR
$20A VSERIN:.RES    2    ;POKEY SERIAL INPUT READY IRQ
$20C VSEROR:.RES    2    ;POKEY SERIAL OUTPUT READY IRQ
$20E VSEROC:.RES    2    ;POKEY SERIAL OUTPUT COMPLETE IRQ
$210 VTIMR1:.RES    2    ;POKEY TIMER 1 IRQ
$212 VTIMR2:.RES    2    ;POKEY TIMER 2 IRQ
$214 VTIMR4:.RES    2    ;POKEY TIMER 4 IRQ
$216 VIMIRQ:.RES    2    ;IMMEDIATE IRQ VECTOR
$218 CDTMV1:.RES    2    ;COUNT DOWN TIMER 1
$21A CDTMV2:.RES    2    ;COUNT DOWN TIMER2
$21C CDTMV3:.RES    2    ;COUNT DOWN TIMER 3
$21E CDTMV4:.RES    2    ;COUNT DOWN TIMER 4
$220 CDTMV5:.RES    2    ;COUNT DOWN TIMER 5
$222 VVBLKI:.RES    2    ;IMMEDIATE VERTICAL BLANK NMI VECTOR
$224 VVBLKD:.RES    2    ;DEFERRED VERTICAL BLANK NMI VECTOR
$226 CDTMA1:.RES    2    ;COUNT DOWN TIMER 1 JSR ADDRESS
$228 CDTMA2:.RES    2    ;COUNT DOWN TIMER 2 JSR ADDRESS
$22A CDTMF3:.RES    1    ;COUNT DOWN TIMER 3 FLAG
;
$22C CDTMF4:.RES    1    ;COUNT DOWN TIMER 4 FLAG
;
$22E CDTMF5:.RES    1    ;COUNT DOWN TIMER FLAG 5
;
$234 LPENH:.RES     1    ;LIGHT PEN HORIZONTAL VALUE
$235 LPENV:.RES     1    ;LIGHT PEN VERTICAL VALUE
;
$244 COLDST:.RES    1    ;COLDSTART FLAG (1=IN MIDDLE OF COLDSTART)
;
$270 PADDL0:.RES    1    ;POTENTIOMETER 0 RAM CELL
$271 PADDL1:.RES    1
$272 PADDL2:.RES    1
$273 PADDL3:.RES    1
$274 PADDL4:.RES    1
$275 PADDL5:.RES    1
$276 PADDL6:.RES    1
$277 PADDL7:.RES    1
$278 STICK0:.RES    1    ;JOYSTICK 0 RAM CELL
```

```
$279 STICK1:.RES    1
$27A STICK2:.RES    1
$27B STICK3:.RES    1
$27C PTRIG0:.RES    1    ;PADDLE TRIGGER 0
$27D PTRIG1:.RES    1
$27E PTRIG2:.RES    1
$27F PTRIG3:.RES    1
$280 PTRIG4:.RES    1
$281 PTRIG5:.RES    1
$282 PTRIG6:.RES    1
$283 PTRIG7:.RES    1
$284 STRIG0:.RES    1    ;JOYSTICK TRIGGER 0
$285 STRIG1:.RES    1
$286 STRIG2:.RES    1
$287 STRIG3:.RES    1
;
$290 TXTROW:.RES    1    ;TEXT ROWCRS
$291 TXTCOL:.RES    2    ;TEXT COLCRS
$294 TXTMSC:.RES    2    ;FOOLS CONVRT INTO NEW MSC
;
$2A3 TABMAP:.RES    15
;
$2B6 INVFLG:.RES    1    ;INVERSE VIDEO FLAG (TOGGLED BY ATARI KEY)
$2BE SHFLOK:.RES    1
;
$2C0 PCOLR0:.RES    1    ;P0 COLOR
$2C1 PCOLR1:.RES    1    ;P1 COLOR
$2C2 PCOLR2:.RES    1    ;P2 COLOR
$2C3 PCOLR3:.RES    1    ;P3 COLOR
$2C4 COLOR0:.RES    1    ;COLOR 0
$2C5 COLOR1:.RES    1
$2C6 COLOR2:.RES    1
$2C7 COLOR3:.RES    1
$2C8 COLOR4:.RES    1
;
$2E4 RAMSIZ:.RES    1    ;RAM SIZE (HI BYTE ONLY)
$2E5 MEMTOP:.RES    2    ;TOP OF AVAILABLE USER MEMORY
$2E7 MEMLO:.RES     2    ;BOTTOM OF AVAILABLE USER MEMORY
;
$2EA DVSTAT:.RES    4    ;STATUS BUFFER
;
$2F0 CRSINH:.RES    1    ;CURSOR INHIBIT (00 = CURSOR ON)
;
$2F4 CHBAS:.RES     1    ;CHBAS REGISTER RAM
;
$2FB ATACHR:.RES    1    ;ATASCII CHARACTER
$2FC CH:.RES        1    ;GLOBAL VARIABLE FOR KEYBOARD
$2FD FILDAT:.RES    1    ;RIGHT FILL DATA (DRAW)
$2FE DSPFLG:.RES    1    ;DISPLAY FLAG : DISPLAY CNTLS IF NON-ZERO
$2FF SSFLAG:.RES    1    ;STRT/STP FLAG FOR CNTL-1;CLEARED BY BREAK
;
;
;
;       PAGE THREE RAM ASSIGNMENTS
;
$300 DCB     =*          ;DEVICE CONTROL BLOCK
$300 DDEVIC:.RES    1    ;PERIPHERAL UNIT 1 BUS I.D. NUMBER
$301 DUNIT:.RES     1    ;UNIT NUMBER
$302 DCOMND:.RES    1    ;BUS COMMAND
```

```
$303 DSTATS:.RES    1    ;COMMAND TYPE/STATUS RETURN
$304 DBUFLO:.RES    1    ;DATA BUFFER POINTER LOW BYTE
$305 DBUFHI:.RES    1
$306 DTIMLO:.RES    1    ;DEVICE TIME OUT IN 1 SECOND UNITS
$307 DUNUSE:.RES    1    ;UNUSED BYTE
$308 DBYTLO:.RES    1    ;NUMBER OF BYTES TO BE TRANSFERED LOW BYTE
$309 DBYTHI:.RES    1
$30A DAUX1:.RES     1    ;COMMAND AUXILIARY BYTE 1
$30B DAUX2:.RES     1
;
$31A HATABS:.RES    35   ;HANDLER ADDRESS TABLE
;
$340 IOCB:               ;I/O CONTROL BLOCKS (IOCB #0 AT $340)
$340 ICHID:.RES     1    ;HANDLER INDEX NUMBER (FF = IOCB FREE)
$341 ICDNO:.RES     1    ;DEVICE NUMBER (DRIVE NUMBER)
$342 ICCOM:.RES     1    ;COMMAND CODE
$343 ICSTA:.RES     1    ;STATUS OF LAST IOCB ACTION
$344 ICBAL:.RES     1    ;BUFFER ADDRESS LOW BYTE
$345 ICBAH:.RES     1
$346 ICPTL:.RES     1    ;PUT BYTE ROUTINE ADDRESS - 1
$347 ICPTH:.RES     1
$348 ICBLL:.RES     1    ;BUFFER LENGTH LOW BYTE
$349 ICBLH:.RES     1
$34A ICAX1:.RES     1    ;AUXILIARY INFORMATION FIRST BYTE
$34B ICAX2:.RES     1
$34C ICSPR:.RES     4    ;FOUR WORK BYTES
;
$350.RES            16   ;IOCB 1 (SAME FORMAT AS ABOVE)
$360.RES            16   ;IOCB 2 (AS ABOVE, ETC.)
$370.RES            16   ;IOCB 3
$380.RES            16   ;IOCB 4
$390.RES            16   ;IOCB 5
$3A0.RES            16   ;IOCB 6
$3B0.RES            16   ;IOCB 7
;
$3C0 PRNBUF:.RES    40   ;PRINTER BUFFER
;
;
;
;       PAGE FOUR RAM ASSIGNMENTS
;
$3FD CASBUF:.RES    131  ;CASSETTE BUFFER
;
; USER AREA STARTS HERE AND GOES TO END OF PAGE FIVE
$480 USAREA:.RES    128  ;SPARE
;
;
;
;       PAGE FIVE RAM ASSIGNMENTS
;
;       PAGE FIVE IS RESERVED AS A USER WORK SPACE
;
;       NOTE: SEE FLOATING POINT SUBROUTINE AREA FOR PAGE FIVE CELLS
;
;
;
;       PAGE SIX RAM ASSIGNMENTS
;
; PAGE SIX IS RESERVED AS A USER'S USER WORK SPACE
```

In addition to the variables listed above, which are supported by all revisions of the operating system, the following are supported by the newer operating systems:

```
$236 BRKKY:.RES     2    ;BREAK KEY VECTOR   (rev. B ff)
;
$26B CHSALT:.RES    1    ;CHAR SET ALTERNATE. (rev. 11 ff)
;
$26E FINE:.RES      1    ;FINE SCROLLING FLAG  (rev. 10 ff)
;
$2D9 KRPDEL:.RES    1    ;AUTO REPEAT DELAY. (rev. 10 ff)
$2DA KEYREP:.RES    1    ;AUTO REPEAT RATE. (rev. 10 ff)
$2DB NOCLIK:.RES    1    ;KEY CLICK DISABLE FLAG. (rev. 10 ff)
$2DC HELPFG:.RES    1    ;HELP KEY FLAG. (rev. 10 ff)
;
$2E9 HNDLOD:.RES    1    ;USER LOAD FLAG. (rev. 10 ff)
;
$33D PUPBT1:.RES    1    ;POWER-UP.... (rev. 10 ff)
$33E PUPBT2:.RES    1    ;....VALIDATION.... (rev. 10 ff)
$33F PUPBT3:.RES    1    ;....BYTES. (rev. 10 ff)
;
$3F8 BASICF.RES     1    ;BASIC SWITCH INTO SYSTEM FLAG (rev. 1 ff)

$3FB CHLINK:.RES    2    ;LOADED HANDLER CHAIN  (rev. 10 ff)
```

Be very wary of using some of these variables. While Atari plans to keep these functions in the OS, future plans may change. Be particularly wary of the functions which are more "systems" and less "user" oriented (HNDLOD, CHLINK, PUPBT1-3, BASICF). On the other side of the coin, if you use the functions supplied by these newer OS revisions, your program will either not work properly if used on older revisions, or your program must test which OS is in use and act accordingly.

From $200 to $3FC, there are no bytes available for applications program use. RAM in this area is used for OS internal variables, or for communications between the user and the OS (e.g., IOCB's). "Spare" or "reserved" areas may be available for your use if your program is used only on one of the OS revisions, but if you intend that your program be used on any Atari computer (even future ones), leave these areas alone. They are reserved for future Atari OS use.

RAM in this area may be "shared" between the user and the OS, as discussed earlier in the page zero section. Follow the same guidelines. You can use the IOCB's without saving their contents if they are not to be used for I/O (the OS uses them only if you OPEN them). The entire area from $200 to $3FF ($200 to $3EC, rev. 10 ff) is cleared to zero and re-initialized whenever SYSTEM RESET is pressed, so your IOCB variables will have to be re-initialized, too. Remember that the OS automatically opens IOCB zero for the screen editor; Atari BASIC uses IOCB's 6 and 7 for graphics I/O and LPRINT.

Similarly, the printer buffer and cassette buffer may be used for your variables if you do not intend to perform cassette or print I/O. Note that the printer buffer, and part of the cassette buffer, are cleared to zero on SYSTEM RESET. The cassette buffer is used by the OS and DOS during the cold-start booting process, but is not used again unless cassette I/O is performed.

The second half of page four, and all of page five, is reserved for use by the Atari BASIC interpreter. This area is not zeroed out on SYSTEM RESET. If your application is in assembly language, all of this area is at your disposal – unless you are using the floating point subroutines. The floating point routines use $580 through $5F2. If your program is co-residing with another language interpreter, or some other program, check if it is using any of these areas before you go ahead and use them.

Page 6 is at your disposal, even with the Atari BASIC interpreter. Page 6 is not guaranteed to be zeroed out when you get it (for example, the 850 interface module uses page 6 during its booting), nor is it zeroed out on SYSTEM RESET. Page 6 is the most well-known free area in the system RAM. If your program will co-reside with other programs (a subroutine package or I/O device driver) check carefully that it does not use page 6 before you go ahead and use it. Some Atari programs use this area (e.g., the CX85 keypad handler), and even more APX programs use this area. If you do not know what programs may be co-loaded with your program, think twice about using page 6 yourself.

Pages 7 and up form the main user area – if DOS or certain device handlers are not loaded. Once again, this area is NOT guaranteed zeroed: AUTORUN.SYS and HANDLERS.SYS are examples of boot-time programs which leave stuff behind. If DOS and/or device handlers are loaded, the actual low memory boundary depends on just what was loaded.

MEMLO defines the boundary between the OS area and the user area. It has the value $700 if no DOS or handler is loaded; it points just beyond the DOS or handler if one is loaded. It is a system convention that the value of MEMLO established by the OS, DOS, and loaded handlers at the end of the power-up initialization will remain unchanged for the remainder of the computing session. There are a couple of "gotchas" here, however. First, there is the configuration byte in DOS 2 which controls the number of FMS buffers (at $709). This byte may be POKEd at any time during a computing session, and the number of buffers DOS believes it has available will change accordingly. DOS 2 will even recompute MEMLO if you change this byte and press SYSTEM RESET (if you don't press [[SYSTEM]] RESET, DOS 2 will still use the buffers, even without changing MEMLO). This is NOT the intended use of this configuration byte, however. When properly used, $709 is changed only immediately prior to writing the DOS FILES to a new DOS disk – then booting from that new disk is the proper way of effecting the new MEMLO from the changed number of buffers. One of the reasons that MEMLO is not changed during a computing session is that programs keep their own copies of MEMLO (unknown to the OS). Atari BASIC is an example. This turns out to be a good thing, since there is a problem with MEMLO computation if both DOS 2 and the 850 interface module are installed and SYSTEM RESET is pressed. In this case, MEMLO is set to reflect DOS 2 but not the 850. It is best, therefore, to do what BASIC does: make your own copy of MEMLO just after cold start, and keep it for the rest of the session. (By the way, BASIC re-copies MEMLO into its own variable when the NEW command is executed, but not on SYSTEM RESET. If you are using the 850 and DOS 2 together with BASIC, avoid using the NEW command after you've hit SYSTEM RESET.)

While MEMLO does not change value during a computing session (at least it's not supposed to), its value CAN change widely from session to session, depending on the configuration. DOS 2 and DOS 3 are roughly the same size, but it is possible that some future Atari DOS will be larger (to take advantage of higher capacity disks, for example). Even the "size" of the same DOS may vary, if the number of buffers is raised or lowered. Then there may be one or more "booted" handlers, such as the 850 interface module RS232 handler. Additionally, through APX and other sources, there are some "utility" programs and subroutines which are loaded from disk and place themselves on the "system" side of MEMLO (thus raising the value of MEMLO).

The high-end memory boundary (MEMTOP) is even more variable; in fact, it IS allowed to change value during a session. The major effect on the value of MEMTOP is the fact that Atari computers can be configured with varying amounts of RAM, from 8K through 48K bytes. Eight-K systems were sold only in the very beginning days, and are probably quite rare. 16K systems are very common, however, and it is a good idea to write your application so that it will work on 16K systems. The amount of RAM does not vary, of course, during a session, but the screen mode can be changed, and different modes take different amounts of RAM for the display area. The display area is always adjacent to the top of RAM. The boundary between the user area (which ends just below the screen memory) and the screen memory is always defined by the OS in MEMTOP. MEMTOP takes lower values when screen modes are used which require more RAM for the display (the largest display mode uses 8K or so).

So how do you use this user area whose boundaries are so shifty? One possible answer is "any way you want", and such anarchy has indeed been common. However, there are some simple things to think about which might help you in making the decision that will be best for your application.

If your program and/or data will be a fixed size (unlike, for example, an editor, where the data size depends on what the user does), then think about the following. First, try to keep the program small enough so that it will run on a 16K system. Even better, allow space for the OS, etc., to grow around your program. $3000 is a very common address to start programs from. This allows plenty of room for possible future larger DOS's, and also allows for most conceivable I/O handler combinations.

If your program has open-ended data requirements, or it is simply a very large program, you may not want to "waste" the space to allow for OS growth. One common alternative is to assume your program will NOT be used with future OS's. For example, you could design your program to work with DOS 2 only (even then, with control of the number of buffers kept out of your user's hands). This is fine, but could be a headache for you in the future, where your program could be "antiquated" by not being able to use all the features of new Atari hardware or software because your program is "tied" to the older software. You could, of course, release another version of your program when that happens.

If you are writing a "utility" (for example, a subroutine package), you may want to put it right after the DOS, so space won't be wasted. This pins down the number of buffers the user is allowed. Better would be to release the source for your utility, so users can assemble the code where they want it. (Or put your utility in page 6, discussed above.)

For these utilities or large programs, where OS "growing space" is to be eliminated, you should also consider making your program relocatable. It is possible (though somewhat tricky) to write truly relocatable code for the 6502 microprocessor, so you code can run properly wherever it is loaded. Or you could write a relocator for your program, which either runs each time the program is loaded or allows the user to choose a relocation address which is used to create a version of your program permanently relocated to that address. If you restrict the relocation to 256-byte intervals (page boundaries) relocation is highly simplified.

If your application will be in a ROM cartridge, relocation may be simpler, since only data (and not code) will be in RAM. Indirect addressing is the key. This is the way most of the Atari programming language cartridges allow for use of all the available RAM, regardless of the specific low and high memory boundaries.

There is a small area between the display memory and the actual top of RAM. The size of this area varies, depending on the graphics mode. It is unlikely that the use of this area will change with future Atari OS revisions, but this is NOT guaranteed. Furthermore, there is a problem with early OS revisions through Rev. B: any time the screen is re-opened (including SYSTEM RESET), not only is the screen memory zeroed, but also some or all of this little spare area. Avoid using this memory.

Upper memory and OS entry points

The area from $C000 to $CFFF is undefined on the original 400 and 800 computers. Beginning with the 1200XL computer (rev. 10 OS), this area contains OS ROM.

$D000 through $D7FF contains memory-mapped I/O, including ANTIC, CTIA/GTIA, the PIA, and POKEY. Newer machines have some new functions added in this area, notably the parallel I/O bus registers. The functions of the various registers in this area are described in the Hardware Manual. This area is reserved for Atari use.

$D800 through $DFFF contains the floating point package, described in the OS User's Guide.

$E000 through $FFFF have always contained the OS ROM. Included in this area are the hardware vectors, 40-character-mode character set, entry point tables, and OS code. The character set and entry tables (described below) are defined for all OS revisions (or just for newer revisions, where so noted). However, the contents of the code areas of the OS are NOT guaranteed, and in fact vary from revision to revision. If your code makes any assumptions about the OS code, either by jumping directly into the OS code, looking at a specific location and expecting what is found there to be invariant in content or meaning, using tables that are not guaranteed, or assuming that the value of a pointer (either in RAM or ROM) which points into the OS code is a constant, will virtually guarantee that your program will not be compatible with all versions of the OS (present or future). That there are a few programs which have violated this rule yet still work is just a matter of blind luck!

Here is the list of supported entry points, vectors, and tables, followed by some notes. Following that will be a list of entries supported only on the newer OS revisions. The intended uses of these entry points and tables are discussed in the OS User's Handbook; for the newer entry points, the User's Handbook 1200XL Supplement.

```
;
; JUMP (OR JSR) VECTORS:
;
$E450 DINITV: JMP     DINIT       ;DISK HANDLER INIT.
$E453 DSKINV: JMP     DSKIF       ;DISK HANDLER ENTRY.
$E456 CIOV:   JMP     CIO         ;CIO ENTRY.
$E459 SIOV:   JMP     SIO         ;SIO ENTRY.
$E45C SETVBV: JMP     SETVBL      ;SET VBLANK PARAMETERS.
$E45F         JMP     SYSVBL      ;SYSTEM VBLANK PROCESSOR.
$E462 XITVBV: JMP     XITVBL      ;EXIT VBLANK PROCESSOR.
$E465 SIOINV: JMP     SIOINT      ;SIO INIT.
$E468 SENDEV: JMP     SENDEN      ;ENABLE SIO XMIT.
$E46B INTINV: JMP     IHINIT      ;INTERRUPT HANDLER INIT.
$E46E CIOINV: JMP     CIOINT      ;CIO INIT.
$E471 BLKBDV: JMP     PUPDIS      ;"BLACKBOARD" (POWER-UP DISPLAY)
$E474 WARMSV: JMP     RESET       ;WARMSTART ENTRY.
$E477         JMP     COLD        ;COLDSTART ENTRY.
$E47A RBLOKV: JMP     RBLOK       ;READ CASSETTE RECORD.
$E47D CSOPIV: JMP     OPINP       ;OPEN CASSETTE READ.
;
```

```
;
; SCREEN EDITOR HANDLER ENTRY POINTS
;
$E400 EDITRV:.WORD   EOPEN-1
$E402.WORD   ECLOSE-1
$E404.WORD   EGETCH-1
$E406.WORD   EOUTCH-1
$E408.WORD   RETUR1-1    ;(STATUS)
$E40A.WORD   NOFUNC-1    ;(SPECIAL)
$E40C         JMP     PWRONA
;
; DISPLAY HANDLER ENTRY POINTS
;
$E410 SCRENV:.WORD   DOPEN-1
$E412.WORD   ECLOSE-1
$E414.WORD   GETCH-1
$E416.WORD   OUTCH-1
$E418.WORD   RETUR1-1    ;(STATUS)
$E41A.WORD   DRAW-1      ;(SPECIAL)
$E41C         JMP     PWRONA
;
; KEYBOARD HANDLER ENTRY POINTS
;
$E420 KEYBDV:.WORD   RETUR1-1
$E422.WORD   RETUR1-1    ;(CLOSE)
$E424.WORD   KGETCH-1
$E426.WORD   NOFUNC-1    ;(OUTCH)
$E428.WORD   RETUR1-1    ;(STATUS)
$E42A.WORD   NOFUNC-1    ;(SPECIAL)
$E42C         JMP     PWRONA
;
; PRINTER HANDLER ENTRY POINTS
;
$E430 PRINTV:.WORD   POPEN-1     ;PRINTER HANDLER OPEN
$E432.WORD   PCLOSE-1    ;PH CLOSE
$E434.WORD   BADST-1     ;PH READ
$E436.WORD   PWRITE-1    ;PH WRITE
$E438.WORD   PSTAT-1     ;PH STATUS
$E43A.WORD   BADST-1     ;PH SPECIAL
$E43C         JMP     PINIT      ;PH INIT.
;
; CASSETTE HANDLER ENTRY POINTS
;
$E440 CASETV:.WORD   OPENC-1
$E442.WORD   CLOSEC-1
$E444.WORD   GBYTE-1
$E446.WORD   PBYTE-1
$E448.WORD   STATU-1
$E44A.WORD   SPECIAL-1
$E44C         JMP     INIT
;
$FCD8         JMP     CLICK       ;ENTRY OR JMP TO KEYCLICK SUBR.
;
$E4C0         RTS                 ;RTS ALWAYS AT $E4C0
```

In addition to the above vectors, the character set at $E000 is guaranteed. Not only will the character set always be at $E000, but its contents will never change; that is, the character set itself is guaranteed.

The 6502 machine vectors (RESET, NMI and IRQ) are, or course, always present. The actual values in these vectors vary from OS revision to revision. Just below these vectors is an ID field, starting with OS revision 11. This was discussed earlier in this document.

Starting with revision 10, the following entry points, etc., have been defined:

```
$E480 PUPDIV: JMP     PUPDIS      ;POWER-UP DISPLAY.
$E483 SLFTSV: JMP     SLFTST      ;SELF TEST.
$E486         JMP     PHENTR
$E489         JMP     PHULNK
$E48C         JMP     PHINIS
```

Revision 10 and all subsequent OS revisions to date also have a second character set, the International ATASCII character set. There are no plans to discontinue support for this character set; however, its existence is not at this time guaranteed. As long as it exists, it will be at $CC00.

The following handler table has been added to support the parallel I/O bus (beginning with rev. 1). From the point of view of CIO, parallel bus device handlers are called in the same way the other device handlers are called; this single handler table serves for ALL parallel bus devices. The parallel I/O bus protocols are described in the appropriate Atari documentation about the parallel bus.

```
; PARALLEL DEVICE HANDLER ENTRY POINTS
;
$E48F GPDVV:.WORD    PPOPEN-1
$E491.WORD           PPCLOS-1
$E493.WORD           PPGETC-1
$E495.WORD           PPPUTC-1
$E497.WORD           PPSTAT-1
$E499.WORD           PPSPEC-1
$E49B       JMP      PPINIT
```

Experience has shown that, in many cases, code which jumps directly into the OS code could have been easily written so that this was not necessary – if only the writer of the code had known better. Given that jumping into the OS generally leads to trouble, here are some ideas on how not to need to do that.

Most of the OS functions you need have defined vectors pointing to the appropriate code. The JMP vectors in ROM have just been listed; the other major group of vectors were shown in the RAM section – they begin at $200. More details on using the RAM vectors appears in the next section. Study these tables of RAM and ROM vectors – they are a good guide to what the OS offers. Pay particular attention to the RAM vectors for interrupt handling.

"Partial" OS functions (essentially subroutines within the OS) do not have defined vectors. Sorry, such subroutines are off limits. The entry address of such routines is not guaranteed (and has probably already changed in existing revisions). Further, the exact function or method of such routines may vary, though this is less likely. Feel free to copy from such routines into your own code – this should be pretty safe as long as the subroutine is pretty well self-contained. If it interacts much with other parts of the system – hardware, other subroutines, the I/O, etc. – then copying it into your code may be more dangerous. You will have to make this evaluation yourself.

One unfortunately all-too-common use of an OS "entry" which has been changed is the key translation table in the keyboard-display handler. The proper way to read the keyboard is to call CIO, using either the E: or K: devices. Common reasons to bypass CIO have been to define keys for which there is normally no meaning, change the meaning of some keys, avoid the "lock up" which occurs if no key is typed or if the typed key does not translate to any ATASCII code, and to avoid the key click. There are preferred techniques for achieving all these goals (except avoiding the keyclick – however, the keyclick is under keyboard control starting with OS rev. 10).

The keyboard may be read through CIO line-at-a-time, with all screen editing supported by the OS for you, by reading the keyboard through E:, the screen editor. Of course, the keyboard is "tied" to the screen when E: is used – the OS has control until the user types RETURN. If you want to do your own screen editing, change the meanings of some keys, or interleave processing with keyboard entry (including allowing processing to continue until a valid key is struck) then you should open two IOCB's: E: to do your screen output, and K: for keyboard input (S: may be used for the screen output if it fits your needs better).

Input through K: is always one character at a time. If you call for a character from K: before the user has struck a key, then K: will wait forever for that key. If you need to interleave processing (or other I/O) with keyboard input, the first thing to do is make sure a key has been struck before calling K: to get it. Keyboard input is handled by the OS interrupt handling – when a key is struck, its POKEY value is placed in CH ($xxx). When K: is called via CIO, no I/O really happens – K: is really just a code translator, translating from POKEY key codes into ATASCII (calling K: also generates the key click). When no key has been struck (since the last key was "read" from CH), CH contains the value 255 ($FF). The K: handler loops forever as long as CH is $FF (it changes via interrupts when the keyboard is struck). So, to avoid this looping, don't call K: until CH changes from $FF. Most commonly, applications which interleave keyboard handling with computation or other I/O simply "poll" CH, that is, check it periodically. K: is called when CH is not $FF. Your code does not need to set CH back to $FF – K: does this for you when you call K: to "read" the keystroke.

When CH contains a POKEY key code, there is still no guarantee that K: will return an ATASCII value to you. This occurs when the user types an invalid key (such as CTRL-6 or any CTRL-SHIFT-key combination – all of these are undefined key combinations), or when the struck key is a key which has meaning to the keyboard handler but does not have an ATASCII value (such as CAPS-LOWR, the inverse video key, or the ESC key). This failure of K: to return when a few "odd" keys are struck may not be a problem to your program, but if you really need to keep going, try putting a time-out timer on the call to the keyboard handler. The recommended technique is to set up the VBLANK timer #2 exit routine (vector CDTMA2, $228) to point to a routine of your own that "fakes" a valid keystroke if K: gets stuck on an "invalid" key. When CH changes from $FF, start timer 2 (via SETVBV) to a value of 3 just prior to calling K: through CIO. If the keystroke was "invalid", K: will not return before the timeout occurs. Your exit routine should then force a value into CH which K: will be able to translate and return – recommended is the CLEAR TAB character (seldom used for anything else), which in POKEY key code form is 172 ($AC). If the key struck IS valid, then K: will return an ATASCII value before the timeout counts down to zero. Note that every "invalid" keystroke is translated into the CLEAR TAB character, which is 158 ($9E) in ATASCII. Thus, your program should ignore the $9E character when it is returned from K:, since this means the timeout occurred. Also note that when K: translates a valid keycode and returns its ATASCII value to your program, the timeout timer is still running. There are two ways to deal with the unwanted TAB CLEAR character it would eventually produce: cancel the timer when K: returns (but do this VERY carefully!) or, better, simply ignore the TAB CLEAR when it appears (by checking for the POKEY key code of $AC and not calling K: if $AC appears). Note that invalid keys produce a double keyclick when these techniques are used.

There are a number of ways you might want to modify the meanings of keys. You might want to suppress certain keys (ignore them); translate keys (have a key produce a different ATASCII value); assign new meanings to keys which are not ATASCII codes (making keys perform a function). These functions are easily added to the K: one-byte-at-a-time method described above, either pre-processing the code in CH while it is in POKEY key-code form, or post-processing after K: returns the ATASCII value normally associated with the keyboard. Whether pre or post processing is best depends on your particular application.

RAM vectors and their use

Many of the vectors in RAM point to code or tables which are used in interrupt processing. Therefore, the value of the vector (all vectors are two bytes) must be valid when the interrupt occurs. If your code is halfway through changing a RAM vector, therefore, be careful that your code cannot or will not be interrupted while you are making the change. For IRQ interrupts, set the I-bit (SEI). For NMI interrupts, use SETVBV (see notes earlier in this document about SETVBV). Some vectors are needed when SYSTEM RESET is pressed (such as DOSVEC). Since there is no way to protect yourself from SYSTEM RESET on the newer computers, your best protection is to change such vectors as little as possible.

All vectors which may be used by the OS are initialized by the OS. Some vectors (such as DOSVEC) are initialized to point to "dummy" or "last-resort" routines (in this case, the blackboard, rainbow Atari logo, or self-test). Other vectors, such as the various interrupt vectors, point to the proper OS routines (such as the appropriate interrupt routines or dummy interrupt routines). Nearly all of these vectors are re-initialized by the OS whenever SYSTEM RESET is pressed. CASINI, DOSINI, and DOSVEC are not re-initialized by SYSTEM RESET; these three vectors are initialized only when power is first applied (cold start).

Your program may have two reasons for changing a system vector: either replacing the system function, or adding to it (keeping the system function and additionally adding yours). If you are replacing an OS function, you must be aware of the other code in the system which might rely on the original OS function. If you code will not co-reside with other programs, including other versions of DOS or other I/O device handlers, then your program is pretty much the whole system and you can code it appropriately. There is still the possibility of your OS replacement having some side effect that could clash with some future OS revision; you can minimize the possibility of this being a problem by having a clear understanding of these guidelines and following them.

If you are just adding code to the OS, there are a few different situations that come up. The first of these is called "chaining." The basic idea is that any number of co-residing programs, I/O handlers, and so on, can "link" themselves to a vector. Then when the vector is used, all of these programs are executed in a "chain." The order in which they are executed depends on the exact chaining method used. When properly implemented, a chain can be open-ended; in other words, no program knows anything about any of the other programs in the chain. Vectors through which you may wish to chain are DOSINI, DOSVEC, CASINI, VIMIRQ, VVBLKI, VVBLKD, and so on.

The basic technique for establishing a chain link is to save the current value of the vector, and then change the vector to point to your own interrupt (or RESET or whatever) routine. Note that you MUST save both bytes of the old value, since there is no way of knowing beforehand what the old value is. (Some programs written for early versions of the OS assumed the initial value of some vectors. This became a problem as soon as the OS was revised – remember that vectors in RAM are initialized to point to code somewhere, not to a constant address somewhere. Furthermore, the vector may already be changed by a co-residing program using this chaining technique!) Now watch what happens when the vector is used. If no other program added itself to the chain after your program, then the vector points to your routine. Each program in the chain, including yours, has the option of running immediately (as soon as it gets control) or running only after the original program (the one pointed to when the value of the vector was saved by your initialization code). To run first, just do your code at the point the vector points to, then JMP (not JSR!) to the place pointed to by the vector value you saved. To let the other program(s) run first, your code should start with a JSR (not JMP!) to the place that saved vector value pointed to.

The actual order in which a chain is executed depends not only on whether each chained program lets the previously found programs run first or last, but also on the order in which the chain is built, that is, on the order in which the co-residing programs get to set up the chain during system initialization. This can become complicated further by the fact that most vectors are reset to their original OS values each time SYSTEM RESET occurs. This means that the chain must be re-built on each RESET. Mechanisms for gaining control of the system on RESET vary depending on the system configuration. One obvious one is the initialization of the applications cartridge by the system. Another method is used for disk-booted software, and this is discussed in the section of this document about working with the DOS products. Care must be taken to study this situation carefully, or your vectored "exit routine" may get lost when SYSTEM RESET occurs, or the chain may be built in a different order when the system is reset than the order which was built at power-up (of course, the order may not matter to you).

If you are chaining to an interrupt vector, the amount of TIME your routine takes should be considered when you are studying the side-effects of replacing or adding to the system functions. If your program is a game, for example, it is unlikely that it will be using the I/O, so the amount of time an interrupt routine takes can be determined by your own needs. For an application like an editor or a language processor, however, your program will have to be able to co-reside with the OS serial and parallel bus I/O processing, and possibly the interrupt processing of other I/O handlers. The best rule of thumb to use is to keep your interrupt code as brief as possible. If you are chaining to the immediate VBLANK vector (VVBLKI) it is particularly important to keep it brief. Consider using the deferred vector (VVBLKD) instead. Also be aware that the new parallel I/O bus allows SIO activity simultaneously with interrupt-driven PIO activity, even with more than one PIO device at a time. In other words, the interrupt load competing with SIO (which is the part of the system most susceptible to being fouled up by too much other interrupt processing) will most likely be getting worse in the future.

Code vectors are also used in situations where your code does not return. DOSVEC is an example of a vector which is called with a JMP rather than a JSR. Occasionally a subroutine vector is used the same way – your routine is called by JSR but it does not return. The most common example of this is the vector DOSINI. DOSINI exists to allow the DOS to initialize itself. Normally DOS returns to the OS when the initialization is complete. However, the AUTORUN.SYS mechanism executes as part of the DOS initialization. With DOS 2.0s or DOS 3, AUTORUN.SYS may include programs which are initialized, run, or both. A program which runs, by definition, does not return. Thus, in this case, the OS initialization is not allowed to complete before the "main program" runs. The proper solution is for your application which runs from AUTORUN.SYS to complete the initialization the OS would have done. Specifically, store a zero at COLDST ($244) and store a one into BOOT? ($9) (yes, the question mark is part of the symbol). Atari does not guarantee that these are the only initialization items which will follow the DOSINI call – in fact, OS revisions from Rev. 10 on are different – but it is unlikely that this code will be changed much. The change made in Rev. 10 is a new method for loading I/O handlers, similar in function to the method used to load the 850 interface module RS232 handler, but different in implementation. The new method does not require the AUTORUN.SYS file (HANDLERS.SYS in DOS 3) supplied with the DOS. Instead, this function is built into the OS. In order to load handlers via this new method (as of this writing, there are no handlers which load via this method) the DOSINI must either return control to the OS to complete initialization, or the AUTORUN.SYS program which does not return to the OS should do a JSR PHREST ($E739) (after MEMLO contains the proper address for any handlers which might load – they'll add their size(s) to MEMLO when they load). If your program does not need any such handlers now or in the future, don't bother with this call. Also be very careful – PHREST does not exist prior to Rev. 10.

With DOS 2 or DOS 3

Atari disk operating systems can be described by their various functional building blocks. Knowledge of this structure is useful in order to understand how the memory is used.

All Atari DOS products have at least two main functional blocks: the file management system (FMS) and the disk utility package (DUP). The FMS is the part of DOS which keeps track of files on the disk, and keeps the disk directory up-to-date. FMS is part of the I/O system, and it is used when your program does I/O to the disk (open, close, read, write, and so on). The DUP is the collection of utilities, such as listing the disk directory, renaming, copying, deleting, loading, saving, initializing, and so forth. DUP is not part of the I/O system but it uses the I/O, such as keyboard-screen to present the menu of choices to the user and read commands. DUP uses FMS to access the disk, just like any other program. In the old DOS I product, of which very, very few are currently in use, these two programs were loaded together when the system was powered up. During initialization, the DOS initialization sets DOSVEC to the start address of DUP. If there is no application cartridge and no disk-based program to run, the OS will immediately run DUP because DOSVEC points there. If there is an Atari BASIC cartridge, the DOS command will cause the BASIC cartridge to go to the program DOSVEC points to; thus the DOS command in BASIC causes DUP to run. Note that since DOSVEC is entered from BASIC via a JMP rather than a JSR, the DUP goes back to BASIC by re-starting the BASIC cartridge.

Atari DOS 2.0s is based on the original DOS. DOS I bugs were fixed. AUTORUN.SYS was modified to actually allow programs to run and/or be initialized (the DOS I version of AUTORUN.SYS would have been better called AUTOLOAD.SYS). Finally, DUP was separated from the FMS and not loaded at power-up. DOSVEC is initialized to point to a small program which is loaded with the FMS, called mini-DUP. The main function of mini-DUP is to load the DUP when DOSVEC is called. In other words, DUP is in memory only when it is needed. DOS 2.0s FMS is significantly smaller than the DOS I FMS+DUP, so DOS 2.0s allows applications much more memory. The drawback is the time needed to load DUP each time it is called upon, rather than having to wait for it to load only at power-up. This method of loading DUP also introduced the necessary evil of MEM.SAV.

DOS 3 is a completely re-written DOS. The FMS is completely new, but it still supports all the same I/O calls as DOS 2.0s (plus some added features). The equivalent of mini-DUP is still around, now called KCP (for keyboard command processor), sometimes called KCP-RES (for resident). The DUP equivalent is now called KCP-OVER (for overlay). DOS 3 has a completely different user keyboard interface from DOS 2.0s, including more prompting, more possible functions, and on-line help. In order to work on systems with only 16K of RAM, this greatly enhanced version of DUP (KCP-OVER) had to be split up – some of its functions (called utilities) are only loaded when called from the KCP menu. KCP-OVER is protected by the a MEM.SAV mechanism similar to DOS 2.0s, but the utilities are not. See the DOS 3 manuals for details.

Both DOS 2.0s and DOS 3 allow the user to configure the number of buffers, corresponding to the number of files which may be simultaneously open. This has the effect of changing the size of the FMS, thus changing MEMLO and possibly affecting your application. If your program cannot stand the possibility that the user might change the number of DOS buffers, your best bet is to release the program on a copy-protected diskette so your program cannot be run with any DOS but the one it is sold with. Also consider mentioning your buffer restrictions, if any, in your user manual.

NOTE: for both DOS 2.0s and DOS 3, the default number of buffers is 3.

Some I/O handlers, such as the 850 interface module RS-232 handler, load themselves automatically at MEMLO, then change MEMLO to include their size (some Atari handlers do not relocate, but they still change MEMLO to include their size). Thus, MEMLO remains the boundary between the OS area (below MEMLO) and the user area (above). Elsewhere in this document problems with MEMLO are discussed, but if the bugs are properly avoided, the MEMLO boundary established when the system initialization is complete should not be changed during a computing session (another way of saying this is that all system handlers which reside below MEMLO must be loaded during system initialization). MEMLO is reset by the OS and by DOS at every SYSTEM RESET. Therefore, any handler or other program which resides below MEMLO must re-establish its size by modifying MEMLO. (In the case of I/O handlers, other initialization is required, too, such as re-establishing the HATABS entry which is zeroed by the OS upon SYSTEM RESET.) Therefore, loaded I/O handlers and such things must get control at every SYSTEM RESET. This is done by the chaining method described above, through the vector DOSINI. Since more than one handler may be booted, relocatable handlers should not set MEMLO to a constant value but should instead add their sizes to MEMLO. Remember than MEMLO must always have an even value, so always add an even number. DOS 2.0s does not add its size to MEMLO – it uses a constant. Therefore, if the DOS is last in the chain, MEMLO will have the wrong value. Therefore, handlers should use the chaining method where the DOS is given control first (the JSR first method).

DOS 2.0s and DOS 3 have slightly different mechanisms for handling the memory overlay when DUP (KCP) is called, but the similarities are greater than the differences. DUP (KCP) is not relocatable. In both cases, the utilities are loaded just above the highest possible buffer address. Therefore, DUP (KCP) does not overlay any part of the FMS. If there are not handlers which have increased MEMLO, DUP (KCP) loads on top of the user area and does not clobber the system. If there is a handler, DUP (KCP) may very likely overlay it. If this happens, the first problem is that the copy utility cannot access the handler that has been overlayed (the attempt simply crashes the system). Then when DUP returns control to the cartridge, the handler is not intact. This is not only a problem should the cartridge attempt to access the handler, but also if SYSTEM RESET is pressed, since the overlayed handler probably has some initialization to do. Of course, all the stuff in the user area under the DUP (KCP) is also clobbered. This is the reason for MEM.SAV.

MEM.SAV is entirely optional under DOS 2.0s. If MEM.SAV is not selected, then the user program is always left clobbered and handlers may be clobbered too. When DUP returns to the cartridge (BASIC) the fact that the program area has been clobbered is signaled to BASIC by DUP's having set COLDST to xxxxx. MEM.SAV always saves the area where the DUP loads. This works fine as long as DUP only uses its own RAM, but for some functions (copy, duplicate, etc.) it is more efficient to use all of RAM. MEM.SAV does not work in this case, and the program area is marked bad by DUP's setting COLDST to xxxxx. At any rate, MEM.SAV should be selected whenever handlers are loaded.

The MEM.SAV option is user-selectable under DOS 3 also, but DOS 3 may go ahead and invoke MEM.SAV anyway if it finds that the KCP will be loaded over anything below MEMLO. The utilities still cannot use the handler that gets overlayed, but the handler will be restored in any case so the cartridge can use it and SYSTEM RESET will work. (By the way, a very complex mechanism is invoked under both DOS 2.0s and DOS 3 so that RESET does not bomb out if DUP (KCP) is sitting where the handler is expected.) The DOS 3 utilities that are loaded independently of the KCP have the same effect as the DOS 2.0s functions which use the rest of memory: the user area is lost.

AUTORUN.SYS (and, with DOS 3, HANDLERS.SYS) allow code to be loaded, optionally initialized, and optionally run during DOS initialization. HANDLERS.SYS is functionally the same as AUTORUN.SYS and runs before AUTORUN.SYS when you are using DOS 3. It was split out from AUTORUN.SYS to make the fact that AUTORUN.SYS can be used for both system and user purposes more apparent and easier to deal with. As a rough guide (by no means required) HANDLERS.SYS will contain Atari files, AUTORUN.SYS user files (both must be merged into AUTORUN when using DOS 2.0s). The AUTORUN.SYS released with DOS 2.0s is a handler poller, used to load the 850 interface module RS-232 handler. HANDLERS.SYS released with DOS 3 contains this same program. Multiple files/programs may be loaded and/or initialized from the same AUTORUN/HANDLERS file – they simply need to be concatenated together in the order they will be loaded/initialized. Of course, only one of the programs can be run – the last one loaded with a run address. The run occurs only after all programs are loaded/initialized, even if the program to be run was near the beginning. See the appropriate DOS technical manual for more details. The handler poller/booter released with the DOS master disks, when used, should precede any other files concatenated to AUTORUN since the handler should boot and set MEMLO properly before user programs boot.

AUTORUN.SYS only loads, initializes, and executes load images (assembly language programs). One common use for AUTORUN.SYS is to start a BASIC program. The best way to do this is to use the APX program named LOAD 'N' GO. This program works by temporarily replacing the screen editor, thus "faking" the RUN command needed to start a BASIC program. This technique does not jump directly into the OS or the BASIC cartridge, unlike some other similar programs lurking around. If you get such a program from your friend, know what it is or think twice before including it with a package you sell.

Things to know if you write an I/O handler

BASIC calls PUT bypassing CIO
Warm start: MEMLO, HATABS, HATABS overflow, DOSINI chaining
List of current I/O device names

Do's and don'ts

DO use OS whenever possible (will avoid incompatibilities of feature loss);
DO use f-keys, and DO use alternate keys too;
DO use the help key, and supply an alternate

Miscellaneous

How to disable the BREAK key
DLI/SIO bug (screen color, keyclick screen color bounce)
"Soft" Cold Start: 850 won't re-boot
Cassette output: write 128 bytes immediately after OPEN
Don't call math pack from immediate VBLANK or DLI (Parallel bus)
Skip columns 1, 2, and 39 for overscan

What to put in your user's manual

Use OPTION to disable the built-in BASIC.
Explicitly tell people to turn off peripherals they don't need.

Bugs

BASIC 16-byte re-write
BASIC CTRL-U, CTRL-R
Keyclick/DLI
SETVBV/DLI

Disk protection do's and dont's

Don't use seek time! If possible, don't use access/transmission time either! Status differences.

Equivalent BASIC/Assembler I/O calling sequences

On the proper use of printers

Data base changes from rev. B to 1200

LOCATION	REV.B -- USE	1200 USE
0000	reserved	LNFLG -- for inhouse debugger.
0001	"	NGFLAG -- for power-up self test.
001C	PTIMOT -- to 0314	ABUFPT -- reserved.
001D	PBPNT -- to 02DE	"
001E	PBUFSZ -- to 02DF	"
001F	PTEMP -- eliminated	"
0036	CRETRY -- to 029C	LTEMP -- loader temp.
0037	DRETRY -- to 02BD	"
004A	CKEY -- to 03E9	ZCHAIN -- handler loader temp.
004B	CASSBT -- to 03EA	"
0060	NEWROW -- to 02F5	FKDEF -- func key def ptr.
0061	NEWCOL -- to 02F6	"
0062	"	PALNTS -- PAL/NTSC flag.
0079	ROWINC -- to 02F8	KEYDEF -- key def ptr.
007A	COLINC -- to 02F9	"
0233	reserved	LCOUNT -- loader temp.
0238-0239	"	RELADR -- loader.
0245	"	RECLEN -- loader.
0247	LINBUF -- eliminated	SPLTM0 -- reserved.
0248-026B	"	reserved.
026C	"	VSFLAG -- fine scroll temp.
026D	"	KEYDIS -- keyboard disable.
026E	"	FINE -- fine scroll flag.
0288	CSTAT -- eliminated	HIBYTE -- loader.
028E	reserved	NEWADR -- loader.
029C	TMPX1 -- eliminated	CRETRY -- from 0036.
02BD	HOLD5 -- eliminated	DRETRY -- from 0037.
02C9-02CA	reserved	RUNADR -- loader.
02CB-02CC	"	HIUSED -- loader.
02CD-03CE	"	ZHIUSE -- loader.
02CF-02D0	"	GBYTEA -- loader.
02D1-02D2	"	LOADAD -- loader.
02D3-02D4	"	ZLOADA -- loader.
02D5-02D6	"	DSCTLN -- disk sector size.
02D7-02D8	"	ACMISR -- reserved.
02D9	"	KRPDEL -- auto key delay.
02DA	"	KEYREP -- auto key rate.
02DB	"	NOCLIK -- key click disable.
02DC	"	HELPFG -- HELP key flag.
02DD	"	DMASAV -- DMA state save.
02DE	"	PBPNT -- from 001D.
02DF	"	PBUFSZ -- from 001E.
02E9	"	HNDLOD -- handler loader flag.
02F5	"	NEWROW -- from 0060.
02F6-02F7	"	NEWCOL -- from 0061.
02F8	"	ROWINC -- from 0079.
02F9	"	COLINC -- from 007A.
030E	ADDCOR -- eliminated	JMPERS -- option jumpers.
0314	TEMP2 -- to 0313	PTIMOT -- from 001C.
003D	reserved	PUPBT1 -- power-up/RESET.
033E	"	PUPBT2 -- "
033F	"	PUPBT3 -- "

LOCATION	REV.B -- USE	1200 USE
03E8	"	SUPERF -- Screen Editor.
03E9	"	CKEY -- from 004A.
03EA	"	CASSBT -- from 004B.
03EB	"	CARTCK -- cart checksum.
03ED-03F8	"	ACMVAR -- reserved.
03F9	"	MINTLK -- "
03FA	"	GINTLK -- cart interlock.
03FB-03FC	"	CHLINK -- handler chain.