"The 68xxx Machines"
The Chatham House Company
RD#1 Box 375
Wyoming, DE 19934

- Address Correction Requested -

- First Class -

The 68xxx Machines

Vol. 1, Issue 1, February 1991 Price: $2.75

The Editor's Thoughts

By James DeStafeno

68xxx was born from an unfulfilled want of men of proven 68XX(X) knowledge and experience. They wanted an unbiased, higher level vehicle that would allow them to exchange ideas.

You who have already met these men in "68 NEWS" and other places, know them to be individuals of uncommon ability. You who have not, will have the pleasure in this and future issues of *68xxx*.

These people and others will explore ideas that here to fore have rarely seen the light of day. The SK*DOS flag will be waved for sure. As will that of MONK, OS-9/K, REXDOS, MINIX and others, and yes maybe even RS-DOS and CP/M, and who knows, MS-DOS might even be seen. Nor is there a limit on languages. I expect "C" to be popular, as well as one or two of the Basics, and of course assembler. Bet we will see compiler action too.

Who have you talked with lately that thinks with a little repair here and there, CP/M would be the

This Issue:

best OS for given situations right into the Twentieth Century? Oh yes..., before dismissing the thought, consider the source. He has had books, articles and papers published on the computer and its uses.

I am excited. Main stream has its place, but the unusual and eccentric is exciting. We have quite a journey before us.

So..., the obligatory bottle of Don Perrier Dom has been "popped" with friends. Glasses have been filled, hoisted and the contents downed with words of "well wishing". Now its time to "walk the walk".

As are other upright creatures, our future will be built on our past. I am proud to consider myself a member of the 68XX(X) community. We of the community have been saying for years we want a 68XXX machine built to meet our needs. We now not only have such a machine, we have three.

One of them relates to the original only on the OS-9 side. To the best of our knowledge it will be shipping its first efforts very soon.

A second machine is really two separate boards. The "word" says the upgraded CoCo board is just now being shipped, while the big brother with the 68XXX chip will be shipping shortly.

Peripheral Technology's machine, offered through "delmar company", has been shipping for some time. This offering is setup to fit the needs of the sophisticated home user.

We have the machine we've been asking for. We know we have operating systems equal to or better then anything any other machine has. Now, to earn the respect of the large home user community, we need software equal to that readily available to the main stream community, MS-DOS.

As we see it, ***68xxx*** has a two fold mandate. It is to encouraging the development of quality software for our machine, and to keep a big area wide open for diverse ideas. If you view it differently, let us hear your points.

The articles inside are in sync with the above. For those that have not yet had the pleasure, Bob van der Poel is the author of quality software. Rush Caley is no body's yes man. Ron Anderson is multi-talented. He and Rush have agreed to write for ***68xxx*** on a long term basis. Peter Stark is the author of the well distributed operating system, SK*DOS. We count ourselves lucky to have enlisted the services of these and other men.

Peter was also the publisher of "68 NEWS". ***68xxx*** has joined forces with him. He will continue writing information updates to SK*DOS and articles, while we will publish them, as well as the efforts of others.

As such, all current "68 NEWS" subscribers will be served by ***68xxx***. Peter will still be available. We at ***68xxx*** hope to continue fulfilling the "68 NEWS" subscriber's needs.

And all you that are getting introductory copies, we accept checks and major credit cards. We need YOUR subscription support and comments to keep this venture going.

Finally, I want to thank all those that have encouraged and helped us to get this vehicle off the ground. We are looking forward to the adventure.

Next Month

The second part of Bob van der Poel's 'Selections in C' will continue. Peter Stark shows us an easy method to format assembly language programs. Rush Caley will have comments to keep our juices flowing and heads thinking. Beginner's Corner by Ron Anderson will continue as well.

If you'd like to submit material to 68xxx, we'd be glad to hear from you. Space is restricted, so short articles and programs are preferred.

In future issues there will be a "Letters Column" and perhaps even a classified ad section. Cost for a classified ad would be a $5 minimum charge for the first 99 characters, and $5 for each additional 99 characters.

Publications like 68xxx cannot exist in a vacuum. Let us know what's on your mind. Your letters and articles are always welcome. If you want your submissions returned, please include an envelope large enough to hold it, along with sufficient postage to cover mailing costs. All correspondence should be sent to: **The 68xxx Machines, RD 1, Box 375, Wyoming DE 19934**

Selections In C

By Bob van der Poel

Welcome to the first of my little attempts to share with you my discoveries in C. Not all of these discoveries will be unique, difficult, or even necessary. So how will I select? Simple; all the topics will be of interest to me. Perhaps a bit autocratic, but it should be eclectic enough so that everyone finds something of interest.

If you have a program of any significant size you will end up with statements which do multiway selections. In Basic this is done with ON..GOSUB, Pascal uses CASE, and C uses SWITCH..CASE. But in C we have another way to accomplish multiway selection, a method I have yet to see described in any detail in any of the standard references. This series of articles will show you how to set up jump tables in C programs, a technique I use to make my own code more readable, shorter and faster.

First, a bit of theory. When the C-compiler creates an executable program it assigns each of the functions in the program an address. This address can be absolute or, in the case of OS9, an offset from the start of the program. Calls to the functions are made via subroutine calls to these functions. However, if we know the address of a function we can call it by its name (the normal method), or by its address (the real subject of this series). Let's review what the primers cover in detail: A call to the qsort() function.

```
#define COUNT 20
main()
{
   int icmp();
   int nums[COUNT];

   .... more code

   qsort(nums, COUNT,
sizeof(int), icmp);

   ....
}
```

In the above fragment the declaration "int icmp()" tells the compiler that icmp() is a function returning an integer. The call to the function qsort() passes the base address of the integer array 'nums', the number of items in the array and the size of each element. It is the last parameter, 'icmp', which is the key to this discussion. Here we are passing the address of a function to another function. Qsort() assumes that the address passed to it is a valid one and that it does the job expected. In this case we have passed the address of a function which compares two integers; but qsort() will accept a function to compare just about anything: longs, doubles or even strings.

Qsort(), if it is written in C, will have declarations similar to the following:

```
qsort(base, num, size, comp)
char *base;
unsigned num, size;
int (*comp)();
```

The first three arguments are pretty straightforward, but have a look at (*comp)(). This strange looking declaration states that 'comp' is a pointer to a function returning an integer. When qsort() needs to compare two items in the array it does so in the following manner:

```
t=(*comp)(base,curr);
```

But what does this have to do with shorter, faster, more readable code? Well, this might not be more readable, but it does mean that we only need one version of qsort(). If we could not pass a pointer to a function, qsort() would have to be rewritten for different types of sorts, or with flags and internal calls dependent on the type of data being sorted--both methods would require significantly more code. This way the qsort() routine can function independently of the type of data being sorted-- a neat trick.

Next month we will continue with this discussion by creating an array of pointers to functions. If you have any comments on this article or suggestions for future ones please drop me a note here at the "The 68xxx Machines" or directly to me at PO Box 355, Porthill, ID, 83853.

Tutor
A Generalized Tutoring Program

By Peter Stark Jr.
Peter A. Stark Sr.

TUTOR is a tutoring program. Its beauty is that it can be adapted to most any subject. In this case it is setup to tutor the use of the Star-K Software System Corp. 68XXX bulletin board (BBS). It got started some time ago when an Australian user asked us how to use the Star-K BBS. Although the BBS is easy to learn, learning via an expensive phone call from Australia is not the best way. We started to write a set of instructions, but soon realized that there must be a better way.

And so was born the TUTOR program. TUTOR does a good simulation of the BBS, so you can sit at your computer and interact with it as though you were actually connected to the BBS. But the program is usable for other applications too; it can be adapted to teach a variety of subjects just by changing the data files which tell TUTOR what to display on the screen, what questions to ask, what answers to accept, and what to do next.

The original TUTOR was written in GWBASIC on a PC clone; it was then hand compiled into 68000 assembly language for running under SK*DOS. This article describes what the system does; you can get the program itself either on disk or by downloading it from the BBS.

The TUTOR system consists of three parts:

- 1. A driver program, called TUTOR. The Basic version is called TUTOR.BAS. It documents the program structure. The 68000 assembly language version is called TUTOR.TXT.

- 2. Two Basic programs which may be useful in preparing Tutor lessons. One, called MAKE.BAS, takes a text file and formats it into a lesson file. The second program, called ANALYZER.BAS, analyzes a series of lesson files to display the linkages between them.

- 3. "Lesson Files", a set of text files, tell the TUTOR program how to proceed. These files tell TUTOR what lesson to teach, what right answers to allow, how to respond to wrong answers, and where to proceed from there. Each of these text files is an ASCII file, which has coded information linking it to other lesson files in the sequence. Since these are standard ASCII files, they can be used on different machines without change.

STRUCTURE OF A LESSON FILE

When TUTOR is first started, it looks for a lesson file called WELCOME. The structure of the WELCOME file is the same as any other lesson file; its only unique characteristic is its name, which means it is the first file to be run. Here is a typical WELCOME lesson file:

```
1  5
2  A FILE1
3  B FILE2
4  A_AND_B FILE3
5   FILE4
6  @ FILE5
7  1
8  Sorry. You didn't read the
   instructions correctly.
9  3
10 Hello, and welcome to the
   TUTOR system. Let us start
11 with a simple uestion: Which
   is the first letter of
12 the alphabet?
```

Only the text within the box is actually the lesson file; we have added the line numbers at the left just to make it easier to describe the various parts of the lesson file. Lets now look at the various parts of the file:

A. THE LESSON AND/OR QUESTION.

The last few lines in the lesson file (lines 10 through 12 in this example) provide the text of the lesson, and/or a question to be answered. In this case, TUTOR starts by displaying the words

Hello, and welcome to the TUTOR system. Let us start with a simple question: Which is the first letter of the alphabet?

Any number of lines can be used here. In order to tell TUTOR how many lines there are, this text is preceded by a line count (the 3 in line 9.)

Typically, the student will be asked a question and expected to respond, as in this example. If you want to go on directly to the next lesson without asking a question, then end the text with the line:

```
*-*
```

Make sure to count this line when you provide the count (on line 9).

B. POSSIBLE ANSWERS.

In response to the question in this lesson, the student will normally be expected to type in an answer. The first part of the lesson file (lines 1 through 6 in this example) lists the kinds of answers we expect, and what TUTOR should do in turn.

The 5 in the first line tells TUTOR that the next 5 lines of the file list specific possible responses the student might make. You can have up to 30 possible response lines (limited by the DIM statement at the beginning of the TUTOR.BAS program.)

The next 5 lines tell TUTOR what action to take with these five possible responses:

```
 Response / Resulting Action
 -------- ---------------------
A        / Go to lesson file FILE1
B        / Go to lesson file FILE2
A AND B  / Go to lesson file FILE3
none (i.e., enter) / Go to lesson
                     file FILE4
Anything else      / Go to lesson
                     file FILE5
```

Note the following:

- 1) Only one of the five possible responses (A) is the right one; in this case, TUTOR will continue with lesson FILE1. But this section also tells TUTOR what to do with several specific wrong answers, in which case TUTOR will go to one of the other lesson files, which can correct the student or provide explanations. If desired, these other files can then loop back to this file and ask the question again.

- 2) The responses in the lesson file are all written in UPPER CASE letters, but TUTOR will accept either upper or lower case letters. Hence the student could answer A or a, B or b, or even "a AnD B".

- 3) The words in a multiple word response within the lesson file (as in A_AND_B) are separated by underlines; these underlines match spaces in the student's reply.

- 4) The symbol @ means "anything else". Normally, the @ response will be last in the list, since TUTOR will not check any other responses below it.

- 5) The fourth entry (which reads "FILE4") starts with a space. This describes what will happen if the student gives no answer, but just presses ENTER.

- 6. If the lesson asks no question (and therefore ends with *-*), then only one answer line should be provided. This line starts with @, and gives the name of the lesson file to go to under all conditions.

C. WRONG ANSWERS

The 1 in line 7 of this example tells TUTOR that it should display the 1 line below it if the student types in a response which does not match any of the responses planned for above.

Line 8 ("Sorry. You didn't read the instructions correctly.") is the reply to give if the student types in something which doesn't match any of the above.

Note this: Since the last listed response (in line 6) was @ (which means "anything else"), it is not really possible for the student to give a reply which does not match any of the five responses listed. So in this particular case, the "Sorry. You didn't read the instructions correctly." reply will never be given, and should have been left blank. (We put it in here just to be complete.)

If TUTOR displays the wrong-answer answer, then it will also repeat the last line of the lesson/question. You can take advantage of this to repeat simple questions without having to retype them. If you do not want to do this, simply make the last line of the lesson/question a blank line.

The MAKE.BAS Program

MAKE.BAS changes text files into lesson files. This is helpful because TUTOR counts lines, and gets confused if the number of lines does not match the count listed on the line above. For example, the sample WELCOME file started out as the following text file:

Hello, and welcome to the TUTOR system. Let us start with a simple question: Which is the first letter of the alphabet?

When MAKE.BAS was run, it asked for the right and wrong answers, and automatically inserted the correct line counts.

The ANALYZER.BAS program:

ANALYZER.BAS is a program which analyzes a series of lesson files to check the structure of the lesson. It prints out a cross-reference list which shows (a) all the files called by any lesson in the course, and (b) all the files which call any lesson in the course.

ANALYZER is written in GW Basic, and so is somewhat limited in its capabilities. The main limitation is that it cannot handle more than about 235 lessons at one time.

HOW TO GET AND USE THESE FILES:

You can get a disk with the TUTOR, MAKE, and ANALYZE programs by sending $2 to Star-K Software Systems Corp., P. O. Box 209, Mt. Kisco, NY 10549; please specify whether you want SK*DOS, Flex, or MS-DOS format. In either case, the disk will contain both the Basic and 68000 Assembly language code, as well as a documentation file which gives much more information than there is room for here. Also included is a full set of lesson files which take you through all the steps of using a BBS. You may also download the programs from the Star-K BBS at (914) 241-3307; the file is called TUTOR.ZIP, and has to be unzipped on a PC or compatible machine. (If you download the programs from the BBS, you obviously do not need a BBS tutorial, but it is there anyway.)

Rush Caley, Live!

I've been thinking about how long I've written about the 6809 CPU, and maybe I've said everything possible I have to say. The trends in what's left of that market have taken the support in directions in which I am not particularly interested. But then I look back on all the years of fun writing off the wall stuff and trying to lift the spirits of the reading public. The calls and letters of interest and thanks tell me I have not wasted my time.

So here I am astride the old silver "D" board I bought back in June of 1980. Prior to that time, I would have defined "cursor" as an expert in four-letter words, and a "user" as someone requiring drug rehabilitation. I'm attached to this machine because it was ultimately responsible for my incredible metamorphosis from exschoolteacher and sometime accountant to a systems analyst for Boeing's Computing Management Organization.

"My interest is in the CUSTOMER"

When Jim DeStafeno called to ask me to write for this fledgling newsletter, I told him that I would probably not have too much to contribute. I never bought a CoCo III because it was not worth the trip to pick it up. So what could I say to those third party entrepreneurs developing a "CoCo IV" or whatever shirt-tail relative it is that they are constructing in the Motorola category? But he says: "Just write...it doesn't even have to be about computers." Now there is a fan!

While I don't want to put any dampers on the creativity and spirit of you who are bringing forth these precocious descendants of the CoCo, I do have some pertinent questions. I don't know a great deal about the development so far, other than the machines have the soul of a Motorola 68XXX chip. But other than that, most of what I have heard tells of varying similarity.

But my primary interest is not in the box. Any compondent company can produce a box. Hundreds already have. My interest is in the CUSTOMER. Every product requires customers, and they do not just materialize at the end of an assembly line. Who will be the targeted audience for these machines? With all their differences, are they targeting the same market? Are they only marketing the box? Will there be software applications bundled or sold with it? Will any be available at all? What will motivate customers to choose one of these machines? These are questions that interest me as a 10 year CoCo veteran and possible consumer for the new generation. So believe me, I'm not a naysayer. I'm just a fussy user with legitimate questions and concerns.

Next time, I will write of more pleasant topics; but this time, I leave you the prophetic words of Niccolo Machiavelli from his treatise "The Prince".

"It must be remembered that there is nothing more difficult to plan, more doubtful of success, nor more dangerous to manage than the creation of a new system. For the initiator has the enmity of all who would profit by the preservation of the old institution and merely lukewarm defenders in those who would gain by the new ones."

Advertisers Index

OS9/68000 PROFESSIONAL SOFTWARE

QUICK ED - Screen editor and text formatter $275.00
A high quality documentation tool and program editor ideally suited to laser printer users. Uses function and cursor keys on any terminal, configurable per user. Microjustifies mixed proportional text. Automatic table of contents generation and user-definable macros and commands. Drives any printer. Ideal for multi-user systems. Available on a 30-day try before-you-buy basis.

FLEXELINT V4.00 - The C source code checker . . $495.00
Flexelint finds quirks, idiosyncracies, glitches and bugs in C programs. 60 options control checking by symbol name or error number. Checks include intermodule inconsistencies, definition and usage of variables, structures, unions and arrays, indentation, case fall-through, type conversions, printf and scanf format string inconsistencies, and suspicious semi-colons. A must for all serious C programmers.

IMP - Intelligent Make Program $250.00
IMP does everything you wished Microware's Make would do, and a great deal more. It is well-behaved, consistent, and extremely flexible. It has a built-in C-like preprocessor and has comprehensive debugging facilities. Rules can be user-defined, and make files for jobs other than assembly-language or C compilation are easily constructed.

DISASM_OS9 - OS-9/68K Disassembler $250.00
This high-speed, three-pass 68000 disassembler can also handle the 68010 and 68020. It intelligently decodes module headers and produces symbol information that can be repeatedly edited and passed through the disassembler allowing iterative disassembly. The system libraries are read to supply symbols.

WINDOWS - C Source Code Windowing Library $250.00
This C source code library package supports multiple overlapping windows displayed on one character-based terminal screen. It supports window headers and footers, and pop-up windows. Windows may be moved, panned, written to while off-screen, etc.

PROFILE - User State Program Profiler $270.00
Designed to profile user-state programs. Profile effectively samples a traced execution building statistical information as it goes. It reads symbol table modules to give a function-by-function account of the time spent during execution. The user may "zoom in" on a function to find a smaller range of addresses where time is being spent.

PAN UTILITIES - C Source Code Utility Set $250.00
Forty useful utilities are supplied in this C source code package. Included are utilities to move files, find files, patch disks, undelete, cross-reference C programs, set and remove tabs, and spell-check documents.

DISK CACHING - High Speed Disk Caching $300.00
Typically speeds compilations, dsaves, deldirs, assemblies by between two and ten times. Can be added to most systems. Features read and write caching with user-settable write delay and buffer sizes. Can be used on hard and floppy disks. Ideal for use on single and multi-user systems alike. Demonstration version available free of charge.

PC9 - MS-DOS to OS-9 Windowing System $350.00
PC9 allows an MS-DOS computer to be used as a terminal to multiple processes on a remote OS-9 system linked by a single serial cable. Each OS-9 process is displayed through a resizable, moveable window on the PC screen. Terminal emulation facilities support uMACS and other screen editors and provide a programmable PC keyboard. Access to PC disk drives is also available through the OS-9 unified I/O system, giving disk capability to ROM based OS-9 systems. A hot key switches between DOS and OS-9 displays.

* delmar co *
Middletown Shopping Center - PO Box 78 - Middletown, DE 19709
302-378-2555 FAX 302-378-2556

SYSTEM IV

The SYSTEM IV is a high performance computer system based on the Motorola 68000 microprocessor operating at a clock speed of 16 MHz and has been designed to provide maximum flexibility and versatility. Microware's Professional OS9/68000 operating system is included with the SYSTEM IV providing an efficient multi-user and multi-tasking environment. This provides the user with a PC for home use, small business applications and a viable low-cost solution for many industrial control applications (imbedded systems). Special requirements (such as midi, sound, A-D/D-A, net-working, etc.) are easily handled with readily available low-cost PC/XT boards which can plug into the SYSTEM IV expansion slots. And, as user requirements change or better special function boards become available, they may be added or replaced at the user's option. Thus, when software requiring multi-media or other new capability becomes a reality, the user will be able to add that capability easily and have the latest technology at his disposal.

The design of the SYSTEM IV is derived from previously successful designs and uses components that have been tested and proven in other systems. SYSTEM IV's uniqueness stems from the ability of its designer and manufacturer, Peripheral Technology, to provide well designed, reliable hardware at a low cost. Further, only the functions necessary to the basic operation have been designed into the mother board. Seven PC/XT compatible expansion slots allow an unrestricted selection of standard PC/XT accessory boards by the user. The user is not locked into any preconceived notions of what is best!

The mother board is a 4 layer XT size board which holds the microprocessor, sockets for up to 4 MBytes of 0-wait state RAM, a battery backed-up clock, 4 serial ports, 1 parallel printer port, a high density floppy disk controller, 7 PC/XT compatible expansion slots, a memory expansion connector to allow an additional 8 MBytes of 0-wait state RAM and the necessary system support chips. The base system includes 1 MByte of on-board RAM, a high density floppy disk drive (3 1/2" or 5 1/4"), either a Hercules monochrome video board or a VGA color video board (800 x 600 x 16), hard disk controller, a 200 watt power supply, an AT style keyboard and Professional OS9/68000. The SYSTEM IV is housed in a mini-PC style case capable of holding up to 5 half-height drives.

The user may install other operating systems. These include CPM, UNIFLEX, MINIX, STARDOS, REX and most any other operating system capable of running on the 68000 microprocessor chip.

To permit access to the largest software base available, an MS-DOS board, the ALT86, will be available shortly as a low-cost option. This board has a V30 (8086) microprocessor running at 10 MHz, includes 1 Meg of 0-wait state RAM, uses the Chips and Technology BIOS, comes with MS-DOS 3.3 or 4.01 and plugs into one of the SYSTEM IV expansion slots. Additionally, an OS9/6809 software emulator will be available soon. This emulator will permit running most OS9/6809 software on the SYSTEM IV.

Base System with Hercules Monochrome Board $1,399.00
for VGA Color Board - add $100.00

For kits, contact Peripheral Technology at 404-984-0742

* delmar co *
Middletown Shopping Center - PO Box 78 - Middletown, DE 19709
302-378-2555 FAX 302-378-2556

Beginner's Corner

By Ron Anderson

Note, this column is a continuation of Ron's column printed in "68 NEWS". Please inquire about back-issues to "68xxx".

- - - - - - - - - - - - - - - - - -

Last time I promised we would talk about disk files. The best method would be to work with the last assember program we used. We'll make it read numbers from an input file and show the total on the terminal. In other words, we will write a data file containing numbers, the last one being zero. Our program will read the file and tell us the sum of the numbers. Then we might branch off and write a program to list a text file to the terminal.

The prerequisite to this lesson is to read section 9 of the SK*DOS manual which talks about file handling and file control blocks. The File Control Block or FCB is nothing magic. In the program we have been working with, we set aside STRBUF to be 30 bytes. An FCB is to be 608 bytes. The DOS uses the first 96 for various functions and the next 256 to hold one sector of data. The last 256 are in the "plan ahead" category since it is planned that perhaps sectors will be made 512 bytes rather than the present 256. The disk operating system actually reads a whole disk sector at a time. When you ask for a character the first time, a sector is read into the FCB. When you ask for the second, the DOS gives the next character in the FCB. When it reaches the end of the sector information, it automatically reads the next disk sector into the FCB. Otherwise, the disk would have to be accessed for each byte read, and it would be very slow.

We will use some SK*DOS calls. All we have to do is to load A4 with the address of the start of our FCB and make the appropriate system call. When working with files it is important to use the SK*DOS built-in error handling mechanism. The 68000 has another register, the Condition Code or Status register. We mentioned it when discussing the zero flag. The Status register also has a CARRY flag. This flag is set when an addition overflows indicating that the operation should carry 1 to the next higher byte or word or long. The carry flag may be set by a software instruction, and the DOS sets carry if there is an error in the GETNAM operation according to the SK*DOS manual. The test BCS means Branch if Carry Set. The program jumps to the error handler if carry is set after the GETNAM operation. The error handler prints the appropriate error message and exits.

Similarly, the SK*DOS manual indicates that the Group B routines, all of which involve reading or writing disks, return with the zero flag set if all went well, and the flag cleared (not zero) if there was an error. The FOPENR and FREAD calls are followed with BNE.S ERROR. In this way, all errors are trapped and your computer won't hang up trying to read a file it couldn't open!

```
* ADD NUMBERS INPUT BY USER UNTIL
                        ZERO IS INPUT
* THIS VERSION READS A DATA FILE
 NAM ADD7
*EQUATES
VPOINT EQU $A000
WARMST EQU $A01E
PSTRNG EQU $A035
PCRLF EQU $A034
OUT5D EQU $A038
DECIN EQU $A030
GETNAM EQU $A023 GET FILENAME
FROM COMMAND LINE INTO FCB
FOPENR EQU $A005 OPEN A FILE FOR
                              READ
FREAD EQU $A001 READ A BYTE FROM
                        THE OPEN FILE
FCLOSE EQU $A008 CLOSE THE FILE
PERROR EQU $A037 PRINT FILE ERROR
                             MESSAGE
PUTCH EQU $A033 PRINT CHARACTER
                        TO TERMINAL
LPOINT EQU 758
START DC VPOINT GET POINTER TO
                     SK*DOS VARIABLES
 MOVE.L A6,A0 SAVE POINTER TO
                             VARIABLES
 LEA WFCB(PC),A4 POINT A4 AT THE
                   FILE CONTROL BLOCK
 DC GETNAM GET FILE NAME FROM CMD
                        LINE TO FCB
 BCS.S ERROR IF CARRY NOT CLEAR,
                              ERROR
 DC FOPENR OPEN THE FILE
 BNE.S ERROR IF ZERO FLAG NOT
                             SET, ERROR
 CLR.W D0 TO HOLD SUM OF ENTRIES
ADD0 BSR.S GETNUM
 LEA STRBUF(PC),A1
 MOVE.L A1,LPOINT(A0)
 DC DECIN
 TST.B D1 IF NOT ZERO, SIGN IS
                             NEGATIVE
 BEQ.S ADD1
 NEG.W D5
ADD1 ADD.W D5,D0
 CMP.W #0,D5
 BNE.S ADD0 IF NOT ZERO GO GET
                              MORE
 LEA MSG3(PC),A4
 DC PSTRNG
 CLR.L D5
 MOVE.W D0,D4
 DC OUT5D
 LEA WFCB(PC),A4
 DC FCLOSE
 DC WARMST
* GETNUM SUBROUTINE
GETNUM LEA STRBUF(PC),A1
 CLR.B D1 SET SIGN FALSE OR
                             POSITIVE
 LEA WFCB(PC),A4 POINT AT FCB
                              AGAIN
GET1 DC FREAD CHANGED FROM DC
                        GETCH ******
 BNE.S ERROR
 MOVE.B D5,D4
 DC PUTCH WRITE DATA TO TERMINAL
 CMP.B #$0D,D6OIS IT A CR?
 BNE.S GET01
 MOVE.B #$0A,D4 IF SO, ADD LF FOR
                             TERMINAL
 DC PUTCH
GET01 CMP.B #'-',D5
 BNE.S GET2
 MOVE.B #$FF,D1 SET SIGN NEGATIVE
 BRA.S GET1  DON'T PUT - IN
                              BUFFER
GET2 CMP.B #$20,D4 SEE WHAT WE
                   SENT TO TERMINAL
 BEQ.S DONGET
 CMP.B #$0A,D4 CHANGE TO CHECK
                             FOR LF
 BEQ.S DONGET
 MOVE.B D5,(A1)+
 BRA.S GET1
DONGET MOVE.B #$0D,(A1)
 RTS
* JUMP TO HERE ON AN ERROR
ERROR DC PERROR
 DC FCLOSE
 DC WARMST
WFCB DS.B 608 FILE CONTROL BLOCK
STRBUF DS.B 30
MSG3 DC.B "SUM IS: ",$04
 END START
```

We have used a branch from several places in the program to the routine labeled ERROR. Note that we do our thing to report the error, close the file and exit to SK*DOS. The user can then correct the problem and try again. You can edit a data file with the editor. Put each value on a separate line (i.e. enter the value followed by CR), and remember to make the last value 0 or else you will get an error "Read Past End of File".

Text files in SK*DOS (our data file done with an editor is a text or ASCII file) use a CR alone to indicate the end of a line. If we were simply to read all the characters in the file and write them out to the terminal, they would all be written on one line. That is, the CR would return the cursor to the left end of the line and the next line's data would be written over the previous data. To avoid this, any program that LISTs a

text file to the terminal has to detect a CR and add a linefeed. The GETNUM subroutine has been altered to do this. See lines 58 to 68 of the program. Note that we changed the test at the label GET2 to cmp.b #$20,D4 rather than D5. We have output the character by placing it in D4 and in the process of the PUTCH routine, D5 is altered, so we used D4 for the test. Then, remembering that when we found a CR we output an LF, we changed the next test to check for LF in D4 to terminate the loop for inputting the current number. We still terminate the number in STRBUF with a CR. Incidentally, we have been using $OD for Carriage return and $OA for Linefeed. We can define those at the beginning of the program and use a name that is more meaningful as below:

```
*EQUATES
VPOINT EQU $A000 SET SK*DOS
                 VARIABLE POINTER
WARMST EQU $A01E DOS WARM START
PSTRNG EQU $A035 PRINT STRING
               TERMINATED WITH $04
PCRLF EQU $A034 PRINT crlf TO
                          TERMINAL
OUT5D EQU $A038 OUTPUT 5 DECIMAL
        DIGITS FROM INTEGER WORD
DECIN EQU $A030 INPUT A DECIMAL
        NUMBER FROM COMMAND LINE
GETNAM EQU $A023 GET FILENAME
       FROM COMMAND LINE INTO FCB
FOPENR EQU $A005 OPEN A FILE FOR
                             READ
FREAD EQU $A001 READ A BYTE FROM
                    THE OPEN FILE
FCLOSE EQU $A008 CLOSE THE FILE
PERROR EQU $A037 PRINT FILE ERROR
                           MESSAGE
PUTCH EQU $A033 PRINT CHARACTER
                      TO TERMINAL
* DEFINE CHARACTER CONSTANTS
CR EQU $OD
LF EQU $OA
FF EQU $OC
```

Now where we had CMP.B #$OD,D4 we can use CMP.B #CR,D4. Again we have gotten more descriptive. CR is a common abbreviation for Carriage Return. LF is common for LineFeed, and FF for FormFeed. If you like, you can spell out these names a little more, but the Assembler only uses the first six characters of a name in it's symbol table, so if you use more than six you must be careful that the first six characters uniquely define a name or label. You could use CARRTN, LNFEED, and FORMFD if you like. These would add to the readability of the program at the expense of increased typing and "wordiness".

Now you have learned how to read a text file and use the data in it. We did cheat and do something the easy way. Our data file terminated with a value of 0 to signal the end of the number list so we stopped reading it when we found the 0. In most cases in reading a text file we don't have a way to anticipate the end of the file. In such cases we try to read beyond the end of the file and get an error 8 (Read Past End of File). Since this is not really an error, we trap that specific case and use it as an end of file mark. If we were simply to open a text file and read it, outputting each character to the terminal via PUTCH, using our trick of adding LF to CR, we would eventually get a file error when we reach the end of file. The ERROR routine would simply check to see if the error is 8, and exit without printing any error message. However, it would print any other error message.

Let's do a LIST program. Our first version will only list proper text files that have each line terminated with a CR and no LF. The source listing of the program LIST is such a file and we can use it to test itself by LISTing LIST.TXT.

```
* PROGRAM TO LIST A TEXT FILE TO
                        THE TERMINAL
 NAM MYLIST
* EQUATES
VPOINT EQU $A000
WARMST EQU $A01E
PCRLF EQU $A034
GETNAM EQU $A023 GET FILENAME
       FROM COMMAND LINE INTO FCB
FOPENR EQU $A005 OPEN A FILE FOR
                             READ
FREAD EQU $A001 READ A BYTE FROM
                    THE OPEN FILE
FCLOSE EQU $A008 CLOSE THE FILE
PERROR EQU $A037 PRINT FILE ERROR
                           MESSAGE
PUTCH EQU $A033 PRINT CHARACTER
                      TO TERMINAL
* DEFINE CHARACTER CONSTANTS
CR EQU $OD
LF EQU $OA
START LEA WFCB(PC),A4
 DC GETNAM
 BCS.S ERROR
 DC FOPENR
 BNE.S ERROR
MAINLP LEA WFCB(PC),A4
 DC FREAD
 BNE.S ERROR
 CMP.B #CR,D5
 BNE.S PUT
 DC PCRLF
 BRA.S MAINLP
PUT MOVE.B D5,D4
 DC PUTCH
 BRA MAINLP
ERROR CMP.B #8,1(A4)  THE ERROR
                    LOCATION IN FCB
 BEQ EXIT
 DC PERROR
EXIT DC FCLOSE
 DC WARMST
WFCB DS.B 608
 END START
```

Assemble this using ASM MYLIST +B and look to see that there were no errors. If OK, ASM MYLIST +L. To use this simple list program you will have to spell out the filename complete with extension:

```
1.A/MYLIST MYLIST.TXT
```

If you have had no errors, it should work first try. If you have the DOSPARAM PL=22, PS=Y (or PL=23 or 24), the listing will stop every screenful and wait for you to hit the ESC key. If not, it will zip right up the screen until it gets to the end of the file. The LIST command that comes with SK*DOS is a bit more complex than this one. It is too bad that the old teletype machines had separate Carriage Return and Linefeed functions. SK*DOS text files have only a CR to signal the end of a line. MS-DOS files use only LF. If you were to assemble MYLIST and redirect the listing to an output file:

```
ASM MYLIST +b >MYLIST.LST
```

You would find the file is written just as it would be output to a printer. That is, it has both a CR and a LF at the end of each line. Listing it with MYLIST will be interesting. Try the exercise and see. Yes, each line has a CR and a LF, but MYLIST adds another LF to the CR. In order to accomodate all the possibilities, MYLIST is going to have to get more complex. If we detect a CR and a LF doesn't follow, we will add a LF. If we detect a LF and a CR doesn't follow we will add a CR. If both are present in EITHER ORDER, we won't do anything. Consecutive CRs have to have LFs added to each of them and vice versa. Then MYLIST will work for listing files or redirected output of word processors, or files copied from MS-DOS or whatever.

```
* PROGRAM TO LIST A TEXT FILE TO
                        THE TERMINAL

 NAM GLIST
* EQUATES
WARMST EQU $A01E DOS WARM START
PCRLF EQU $A034 PRINT CRLF TO
                          TERMINAL
GETNAM EQU $A023 GET FILENAME
         FROM COMMAND LINE INTO FCB
```

```
DEFEXT EQU $A024 SET A DEFAULT
                        EXTENSION
FOPENR EQU $A005 OPEN A FILE FOR
                              READ
FREAD EQU $A001 READ A BYTE FROM
                      THE OPEN FILE
FCLOSE EQU $A008 CLOSE THE FILE
PERROR EQU $A037 PRINT FILE ERROR
                            MESSAGE
PUTCH EQU $A033 PRINT CHARACTER
                       TO TERMINAL
* DEFINE CHARACTER CONSTANTS
CR EQU $0D
LF EQU $0A
START LEA WFCB(PC),A4
 DC GETNAM GET FILENAME FROM
                     COMMAND LINE
 BCS.S ERROR
 MOVE.B #1,D4  SET DEFAULT
                   EXTENSION .TXT
 DC DEFEXT
 DC FOPENR OPEN FOR READ
 BNE.S ERROR
MAINLP LEA WFCB(PC),A4 POINT AT
                              FCB
 DC FREAD READ A CHARACTER
 BNE.S ERROR
 CMP.B #CR,D5 IS IT CR?
 BEQ.S FNDCR IF SO GO HANDLE IT
 CMP.B #LF,D5 IS IT LF?
 BEQ.S FNDLF IF SO GO HANDLE IT
 MOVE.B D5,D4 ELSE OUTPUT
            CHARACTER TO TERMINAL
 DC PUTCH
 BRA.S MAINLP GO GET ANOTHER CHAR
FNDCR DC PCRLF OUTPUT CRLF
 DC FREAD GET NEXT CHAR
 CMP.B #LF,D5 IS IT LINEFEED
 BEQ.S MAINLP IF SO SKIP IT
 CMP.B #CR,D5 IS IT CR?
 BEQ.S FNDCR HAVE TO HANDLE
       CONSECUTIVE CR WITHOUT LF
 MOVE.B D5,D4 ELSE OUTPUT IT
 DC PUTCH
 BRA.S MAINLP GO GET MORE
*SAME AS FNDCR ONLY IN REVERSE
FNDLF DC PCRLF
 DC FREAD
 CMP.B #CR,D5
 BEQ.S MAINLP SKIP IT
 CMP.B #LF,D5
 BEQ.S FNDLF HAVE TO HANDLE
       CONSECUTIVE LF WITHOUT CR
 MOVE.B D5,D4 ELSE OUTPUT IT
 DC PUTCH
 BRA.S MAINLP
ERROR CMP.B #8,1(A4)  THE ERROR
                   LOCATION IN FCB
 BEQ.S EXIT IF ERROR = 8 THEN END
                          OF FILE
 DC PERROR
EXIT DC FCLOSE
 DC WARMST
WFCB DS.B 608
 END START
```

Now assemble this one and then assemble it to a .lst file. Finally try 1.A/MYLIST GLIST.LST. You will see that it prints double spaced to the screen. Now try 1.A/GLIST GLIST.LST. It will list normally. I don't have an easy way of testing the other possibilities, but you can see that the handling of the situations is "symmetrical" so the probability that the other two conditions will be handled correctly is pretty high. The only other improvement we could make is to select a "default extension". Most of the files we list will have the extension .TXT. Right after the DC GETNAM line, add the following two lines:

```
MOVE.B #1,D4
DC DEFEXT
```

You will have to add to the equates:

```
DEFEXT EQU $A024
```

Now if the file is .TXT and in the current working directory, you only have to specify the filename as: 1.A/GLIST GLIST. If you want to list a file with a different extension, simply use it: 1.A-/GLIST GLIST.LST.

Well, there you have a fully capable SK*DOS utility program lacking only the HELP facility of the real one that comes with SK*DOS. (Type LIST ? CR and read the help information). We could easily duplicate that as well. Perhaps we'll do that next month.

Perhaps this is enough for this time. I have one warning to give you. If you decide to play with opening a file for write, and writing to it, be very careful that you actually open a file before you write to it. Trap the error as we have done here on FOPENW. Early versions of SK*DOS don't actually check to see if you have a file open for write when FWRITE is called. If a file is not open, you could write anywhere on your disk, but the most likely place is track 0 where the directory resides. Doing this will damage your disk and you may have to reformat it to use it again. Experiment with an empty disk for the output file. DON'T to write to a file on your hard disk with an "undebugged" assembler program unless you have SK*DOS version 3.1 or NEWER. Debug your program by making a floppy drive the working drive. When it all works, only then transfer to the hard disk. I SPEAK FROM EXPERIENCE, HAVING CRASHED A HARD DISK TWICE BY ACCIDENTALLY TRYING TO WRITE TO A FILE THAT HAD NOT BEEN OPENED!

By now the SK*DOS manual should be making a lot of sense to you. Please READ it and then read it again. Soon you will be inventing new utilities for yourself.