The 68xxx Machines

Vol. 1, Issue 2, March 1991 Price: $2.75

The 68xxx Machines is published and copyright (C) 1991 by Catham House Company, RD#1 Box 375, Wyoming, DE 19934. Ph. (302) 492-8511. The editor is James H. DeStafeno. One year USA subscription is $12.50. Canada and Mexico $14.25. All others (surface) $17.25. All major credit cards accepted. Our low prices reflect a 10% cash discount. Please add 10% to credit card orders. Any size display advertising is accepted. The half page rate is $10/issue. Write for other size/duration rates.

Readers are encouraged to contribute letters, articles, programming information and other material related to computers with the 68xx(x) processors; excepting Macs and Amigas. Please send material to the above address. Thank you for your support.

The Editor's Thoughts

By James DeStafeno

Mile stone number one passed by with noteworthy success. No one has told me issue number one stinks. No subscriber of "68 NEWS" asked for their money back. (Because of that Peter Stark has OKed us to carry the colors until farther notice.) Credit card numbers have been phoned in and checks have been mailed in, both asking for a subscriptions. That is our life-line and we thank those that have done so.

While handing out thanks, we have many to go around. Of course I feel blessed to have friends that think enough of me to encourage the project. So "thanks" to all of you.

Randy Krippner is the layout designer. He put in a lot of time, effort and a few bucks of his own to get the result we see. "Thank you", Randy.

I feel lucky to have chosen the printer I did. As you can see their quality is really good and their cooperation is excellent.

This Issue:

"Thank you", Mike and your whole team.

The fact is we have had nothing but praise "for the first effort", save "more pages and writers would be nice". Of course our reply is, "Thank you. Hang in, that's what we want too. We'll get there."

An eagle eye on this month's masthead will notice an increase in the subscription rates. I feel bad raising the rates after only one issue. However, as you know the postage rates went up on the third of February. We were already as close to the bone as we could get. All of you that subscribed from the first issue got in just under the wire.

Speaking of dollars, Randy tried to do Bob van der Poel a favor by changing his S&H rate in his ad from $2 to $20. Bob told us to change it back; "It didn't fool anyone."

I hope you noticed the clear tape we use to hold the pages closed during shipping peels off without tearing the page. Interesting stuff.

I was hoping to have a "Reader's Letters" column this time, but in trying to catch up there has been little time between the mailing of the first issue and going to press for this second issue. Most all the letters we've gotten have a check for a subscription and a note much like Alen Gordon's of Miami, Florida, "Many thanks for the sample issue of <68xxx> you sent." "Best of luck." Maybe next time we'll have enough letters to make a column. If you let us know your thoughts, maybe one of them will be yours.

On the subject of writing, please remember that I am looking for YOUR feed back. The effort is being made for YOU and I want to do good by you. Let me know what you are thinking about. Of course we are always looking for article contributions. What did you just learn and/or would like to share? It doesn't have to be a program, maybe even shouldn't be. Just some experience that gave you satisfaction to get right or completed.

Don't miss the new Classified Ads column. Do you what or have to sell, new or used software or hardware. The cost is low and we'll do the type setting. A "Wanted" ad, $2.50 per 50 character line per issue. A "Sale" ad, $5.00 per 50 character line per issue.

* * * * * * * * * * * * * * * * * *
* God bless our troops in the *
* Gulf and all around the world.*
* * * * * * * * * * * * * * * * * *

Selections In C

By Bob van der Poel

We started our discussion of multiway selections in C last month by reviewing the concept of a pointer to a function. This month we'll get a bit more fancy (certainly not tricky) and expand on this concept by setting up an array of function pointers.

In my new text editor (VED for OSK) I needed a method of creating a series of cursors, depending on the current state of the options. For example, I want a block cursor for insert mode and an underline for overstrike. Complicating things even more, I have to be able to do this in normal and reverse video.

The basic technique to get a keypress while displaying a cursor is pretty straight forward:

1. Set the terminal position to the correct x/y position,
2. Turn on the correct video attribute,
3. Display the character under the cursor,
4. Wait for a keypress,
5. Restore the original video attribute,
6. Return the keypress to the calling function.

In the above, it is (2) and (5) which are the problem: how does the cursor routine know which attributes to use.

The first part of the answer lies in the parameters passed to the function. I pass the x/y screen position, the character under the cursor and the cursor type. The cursor type can be:

- Ø - underline cursor
- 1 - reverse video block (for use on normal video text)
- 2 - normal video block (for use on reverse video text)

For an underline cursor step (2) becomes a call to a function which turns on underlining and (5) turns underlining off. Similar sequences apply to reverse video, etc. Have a look at the following fragment to see how I did it:

```
curkey(x,y,c,ctype)
int x,y;      /* x/y position
    */
char c;       /* character curr
      ently at cursor pos */
int ctype;    /* cursor type Ø=
      UL, 1 & =BLOCK */
{
   register char k, getkey();
   extern int revon(), revoff(
       ), undlnon(), undlnof
       f();
   static int (*curfn[][2])()=
       {
      undlnon,undlnoff,   /*
             overstrike mode

             */
      revon,revoff,       /*
            insert mode

             */
      revoff,revon        /* r
            everse video input
            lines */
   };
   gotoxy(x,y);
   (*curfn[ctype][Ø])();
   writel(c);
   k=getkey();
   gotoxy(x,y);
   (*curfn[ctype][1])();
   writel(c);
   return k;
}
```

There are a number of things to examine here. First, the line starting with 'extern' declares the video functions we use. They need to be declared as external since the actual functions are contained in a separate source file. The next section creates an array which is pointers, pointing to functions. The first element of each entry in the array is the function which enables the necessary attribute, the second disables it. The 'static' modifier is needed so the compiler can set the pointers in the program's data area.

Now examine the actual function: eight lines of fairly cryptic code. The first line sets the correct x/y position. The second line calls a function. Which one? It is the one pointed to by the subscript subscript 'ctype'. The second subscript, Ø, points to the enable routine. If 'ctype' has a value of Ø the function undlnon() will be called. The third line displays the character which is supposed to be under the cursor.

Line four calls a function which gets a single keypress and assigns the result to the variable 'k'. Next the x/y position is reset, the video attributes are restored and the original character is again displayed. Finally, the key is returned to the caller.

This could have been done with SWITCH..CASE statements, but the method used here is much shorter (and faster). Not only that, but it is very easy to change the video attributes used for the cursor and add other cursor types: just change or add more entries to 'curfn[][]'.

Have a good look at the declarations used in this function. I know it looks a bit complex with all those parentheses--but that is the correct (and only) way to do it. My actual function is even shorter than the above one since I wrote another function which sets the x/y position, attribute and character. See if you can set up the correct parameters for this second function and the call to it.

Next month we will expand this concept a bit more and create a jump table using a structure. Until then, 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.

Advertiser's Index

Hand Compiling The Easy Way

By Peter Stark

Although there are several compilers available which run under 68000 SK*DOS, there is none for Basic, a language which I particularly like. And so I frequently write and debug programs in Basic (which is very easy and fast to use), and then hand-compile them into 68000 assembly language.

Although compiling a program by hand seems like a difficult job, actually it is quite easy and fast -- once you get the knack and develop some fairly simple steps to follow. In this article, I will describe how it's done, and give a simple example. Hand-compiling programs will work for almost all Basic programs except those which use floating-point arithmetic. We do not yet have a good series of assembly language floating-point routines. One great advantage of hand-compiling is, if you are reasonably proficient at assembly language programming, you can produce very compact and fast code, probably faster than a compiler might produce.

I have used the procedure several times. For example, I used this method to produce EDLIN, a simple line editor which is part of the SK*DOS operating system. The entire EDLIN program took a weekend to write - one day to write the entire Basic program and debug it, and a second day to translate it to assembly language, and get all the bugs out of it. The procedure I follow goes like this:

1. Write the original program in Basic and debug it completely. Make sure all parts work, and they do what you want them to do. Although it is possible to make changes later, it is easiest to get it all working from the very beginning.

2. Make a copy of the Basic program file (in ASCII, if you work on a PC clone), and rename it with a .TXT extension; this will be the framework of the assembly language program.

3. Using an editor, insert an

asterisk before every line of the program. In this way, the Basic program will become a series of comment lines in the assembly language program. These Basic program lines, plus any other comments which you may add, will provide documentation, which is probably much better than most programs written directly in assembly language. This method describes not just what each part of the program does, but also HOW it does it.

4. Now make a firm rule: do not remove any of these Basic statements, and do not move them around. Keep them in the exact same order as they were in the original Basic program. This will force you to write the assembly language program in the same order.

5. Now look at the beginning of the program. If there are any DIM statements, or any initialized variables; set up the storage for them first. My own preference is to put all storage at the end of the program. Although this separates the Basic declarations (which are usually at the beginning) from the assembly language storage (which is at the end), it seems to keep things more organized.

6. Now continue down the program, and look at each line. If possible, translate each line into assembly language code literally, without even thinking of the context. Sometimes it will be easier to translate an entire group of lines, but wherever possible, treat each Basic line separately. As each line is translated, assume that all data it needs is in memory. Do not carry anything in CPU registers between lines. Each block of code should take all its data from memory, process it, and then return everything to memory when done. While this may make the program a few lines longer, we 68K users usually have so much memory available that we can afford to waste a few bytes here and there. And it does make it much easier to keep track of what is happening in each line.

7. Each block of assembly language code should have a label based on the line number of the Basic line it replaces. For example, the block of code which implements line 6Ø of the Basic program gets the label L6Ø, (where the L stands for Line).

8. I usually make all numeric variables into long integers (using four bytes), and give all strings a maximum length of 79 characters. I use a CR ($ØD) delimiter to mark the end of a string, so I reserve 8Ø bytes for them.

9. If the Basic program uses functions, I usually include similar subroutines in my programs. My assumptions for subroutines are exactly the same as calling SK*DOS functions, namely (a) AØ-A4 and DØ-D4 will never be changed by subroutines, (b) A5-A6 and D5-D7 will be changed, (c) input into a subroutine will be in A4 and/or D4 if possible, and (d) returned results from a subroutine will be in A5 and/or D5, if possible.

Let's look at the following simple Basic program as an example:

```
1Ø FOR I = 1 TO 3
2Ø     INPUT A$
3Ø     L = LEN(A$)
4Ø     PRINT RIGHT$(A$,L-1);
5Ø     PRINT LEFT$(A$,1);
6Ø     PRINT "AY"
7Ø NEXT I
8Ø STOP
```

This program translates three words to what kids sometimes call Pig Latin. The rule for Pig Latin is that you put the first letter of every word on the end of the word, and add "ay" to the end of the word. For example, TABLE becomes ABLETAY. Let's look at the program line by line. First, I start with the usual -- a comment, a library call to bring in SK*DOS function names, and a START line with the version number:

```
* Demonstration program to show
  hand compiling Basic:

       LIB SKEQUATE

START  BRA.S L1Ø
       DC.W $ØØØ1  version number
```

Now I start off translating each line of the program. Line 1Ø becomes:

```
*1Ø FOR I = 1 TO 3
L1Ø       LEA     I(PC),AØ
          MOVE.L #1,(AØ)       I = 1
          LEA     LASTI(PC),AØ
          MOVE.L #3,(AØ)  LAST I=3
```

The above shows that I simply place a 1 into a long integer called I, and a 3 into an integer called LASTI. (Variable storage is all at the end of the program.) Note how this block of code begins with the label L1Ø, and how we use PC-relative addressing to refer to memory. Note also how the original Basic line becomes the comment.

The next line becomes:

```
*2Ø INPUT A$
L2Ø      LEA     QUESTM(PC),A4
         DC      PSTRNG
                       print "? "
         DC      INLINE
                     input string
         LEA     LINBUF(A6),A4
              move "from" address
         LEA     ADOLR(PC),A5
                move "to" address
         BSR.L   STRMOV
               put string into A$
         BRA.S L3Ø
QUESTM   DC.B    "? ",4
```

Since Basic always prints a question mark before the doing the INPUT, we duplicate this here by setting up the appropriate string (QUESTM) and using SK*DOS's PSTRNG function to print it. (Since we're using the SK*DOS print-string function, we use the normal delimiter of 4 to end the string.) The rest of the code uses the SK*DOS INLINE function to input a line of text, sets up A4 to point to the string just read, A5 to point where we want it to be moved into, and then calls the STRing-MOVe subroutine (shown later) to move the string from the line buffer into ADOLR. ADOLR is an 8Ø-character buffer (defined at the end) which holds the string A$.

The following line determines the length of the string A$:

```
*3Ø L = LEN(A$)
L3Ø      LEA     ADOLR(PC),A4
         BSR.L   LEN
         LEA     L(PC),A4
         MOVE.L  D5,(A4)
                     store length
```

We simply point A4 to the string and call the LEN function (shown later). The length, which is returned in D5, is placed into the long integer called L.

The next Basic line prints the right L-1 characters of the string A$. For example, if A$ is "TABLE", then the line prints ABLE:

```
*40 PRINT RIGHT$(A$,L-1);
L40        LEA      ADOLR(PC),A4
                                   A$
           LEA      TEMP(PC),A5
                                 TEMP
           MOVE.L   L(PC),D4
           SUB.L    #1,D4
                                  L-1
           BSR.L    RIGHT
           LEA      TEMP(PC),A4
                                 TEMP
           BSR.L    PRTSTR
                         print string
```

Our assembly code does this in two steps: first, it moves the right L characters into a temporary string called TEMP, and then it prints TEMP. The first part is done with the RIGHT subroutine (which needs A4 to point to the input string A$, A5 to point to the output string TEMP, and D4 to contain the length). The second part is done with PRTSTR (which needs A4 to point to the string). There would have been an easier way of doing this (pointing A4 into the middle of A4 and then using PRTSTR), but I wanted to show how to use the RIGHT subroutine.

The next Basic line prints the first letter of A$. Again, there would have been an easier way (by simply picking up the first character of ADOLR and using SK*DOS's PUTCH function to print it), but I wanted to show the LEFT subroutine:

```
*50 PRINT LEFT$(A$,1);
L50        LEA      ADOLR(PC),A4
                                   A$
           LEA      TEMP(PC),A5
                                 TEMP
           MOVE.L   #1,D4
                                    1
           BSR.L    LEFT
           LEA      TEMP(PC),A4
                                 TEMP
           BSR.L    PRTSTR
                         print string
```

The next Basic line prints the string "AY" at the end of the word. As before, there would have been an easier way, but this example shows how to use PRTSTR:

```
*60 PRINT "AY"
L60        LEA     AYSTRN(PC),A4
           BSR.L    PRTSTR
                   print string "AY"
           DC       PCRLF
                   follow with CR/LF
           BRA.S    L70
                         then continue
AYSTRN     DC.B     'AY',$0D
```

Next we have to implement the NEXT I statement. It takes just a few lines to get I, increment it, put it back, and check against LASTI. As long as I is equal to LASTI or less, we simply go back to line 20:

```
*70 NEXT I
L70        LEA      I(PC),A4
           MOVE.L   (A4),D7
                                    I
           ADD.L    #1,D7
                            increment
           MOVE.L   D7,(A4)
                              restore
           CMP.L    LASTI(PC),D7
           check against last value
           BLS.L    L20
             continue if I <= LASTI
```

Line 80 is almost trivial:

```
*80 STOP L80        DC WARMST
```

then stop

Finally, we have to put in the subroutines. If you are going to do hand-compiling often, it is very useful to write a stock set of subroutines to mimic the standard Basic string functions. I always assume that all strings have a maximum limit of 79 characters plus a delimiter (this is easily changed for special cases). I have done enough debugging of the original Basic source program that I don't have to worry about string overruns etc., so these subroutines are fairly simple. Here the are:

```
* STRMOV subroutine - move a
string from (A4) to (A5). ASSUME
THAT
* ALL STRINGS HAVE LENGTH 80, so
don't bother to check for end
STRMOV     MOVEM.L A4-A5,-(A7)
                        push on stack
STRMO1     MOVE.W   #79,D7
                               80 - 1
STRMO2     MOVE.B   (A4)+,(A5)+
           DBRA     D7,STRMO2
              repeat until 80 moved
           MOVEM.L (A7)+,A4-A5
                                 pull
           RTS
* LEN subroutine - enter with A4
pointing to string, exit with
length in D5 LEN        CLR.L D5
           MOVE.L A4,A5 LENA
```

OS9/68000 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.

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.

SYSTEM IV COMPUTER

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 (embedded 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 improved 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.

TO ACCESS 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, has a socket for an 8087 math co-processor and plugs into one of the **SYSTEM IV** expansion slots. Additionally, an OS9/6809 software emulator/interpreter will be available soon. The emulator/interpreter will permit running most COCO OS9/6809 software on the **SYSTEM IV**.

OTHER OPERATING SYSTEMS may be installed. These include CPM, UNIFLEX, MINIX, STARDOS, REXDOS and most any other operating system capable of running on the 68000 microprocessor chip.

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, 2 parallel ports, a high density (37C65) floppy disk controller, 7 PC/XT compatible expansion slots, a memory expansion connector to allow an additional 6 MBytes of 0-wait state DRAM, keyboard connector and the necessary system support chips.

THE TERMINAL SYSTEM includes the mother board with 1 MByte of on-board DRAM, a high density floppy disk drive (3 1/2" or 5 1/4"), 4 serial port connectors, a parallel printer port connector, a 200 watt power supply, mini-PC style case capable of holding 5 half-height drives and Professional OS9/68000. This configuration requires the use of an external terminal(s).

THE CONSOLE SYSTEM adds a VGA (800 x 600 x 16) graphics board and an AT style keyboard and provides full graphics capability at the console. Terminals may be added.

THE **SYSTEM** IV comes with a one (1) year parts and labor warranty.

TERMINAL System	$ 999.00
CONSOLE System	$1,149.00

OPTIONS

3 MByte additional DRAM		$120.00
Hard Disk Controller and driver		$ 69.00
40 MByte Hard Disk		$295.00
20 MByte Hard Disk		$240.00
Additional 5 1/4" or 3 1/2" HD Floppy Drive		$ 92.00
AT Style keyboard and 800 x 600 x 16 VGA Card and driver		$159.00
For 1024 x 768 x 256 VGA Card w/1 Meg of Memory in place of standard VGA card	add	$170.00
Mono Display Card in place of VGA card	deduct	$ 50.00

Prices subject to change without notice.

Special monitor prices when ordered with the **SYSTEM IV**.

See the PERIPHERAL TECHNOLOGY AD for kits.

* delmar co *

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

```
CMP.B #$ØD,(A5)+
                is next char a CR?
        BEQ.S LENEX
                        yes, exit
        ADD.L #1,D5
             no, increase length
        BRA.S LENA LENEX    RTS
                     finally exit
* LEFT subroutine - put left D4
chars of (A4) string into (A5)
LEFT    BSR.S    STRMOV
        copy entire string first
        ADD.L    D4,A5
        MOVE.B   #$ØD,(A5)
           cut off end of string
        RTS
* RIGHT subroutine - put right D4
chars of (A4) string into (A5)
RIGHT   MOVEM.L A4-A5,-(A7)
                    push on stack
        BSR.S    LEN
       get length of (A4) string
        ADD.L    D5,A4
           point A4 past string
        SUB.L    D4,A4
     then back up to 1st char of
                          desired
        MOVE.L   4(A7),A5
             restore original A5
        BRA.S    STRMO1
      move string, pull, and RTS

* PRTSTR subroutine - print
string pointed to by A4 PRTSTR
MOVEM.L A4-A4/D4-D4,-(A7)
                             push
PRTST1  MOVE.B   (A4)+,D4
                  next character
        CMP.B    #$ØD,D4
                       is it CR?
        BEQ.S    PRTST2
                    yes, so stop
        DC       PUTCH
                 no, so print it
        BRA.S    PRTST1
          then go back for next
PRTST2  MOVEM.L (A7)+,A4-A4/D4-D4
                             pull
        RTS
```

We are almost done. We now need to set aside space for all variables. This data area assumes that all numeric variables are long integers, and that all strings get 8Ø bytes:

```
I       DS.L     1
                                I
LASTI   DS.L     1
             last I in FOR loop
L       DS.L     1
                                L
ADOLR   DS.B     8Ø
                               A$
TEMP    DS.B     8Ø
                temporary string
```

Since Basic assumes all variables are zero and all strings are empty when it starts a program, you may want to initialize this area with appropriate DC statements rather than just defining storage with DS. I am usually careful in my Basic program to assume no initialization, so I do not bother with this extra step.

Finally, we end off the program with

```
        END      START
```

The trick is to play dumb. Don't try to combine lines, and don't try to consider what each Basic line is doing. Simply do what a real compiler does: translate each line as you go and ignore its context. Remember the rules for register usage, both as to what registers subroutines use and/or change, and also the rule that nothing gets left in the 68ØØØ's registers between Basic lines. If you want to make the program smaller or faster, then wait until it is finished and running before trying to optimize. My experience has been that you will not feel the need to make any changes.

So do try it. I have hand compiled several programs from Basic, and found it very useful. In a few cases, where I was stuck on a complex algorithm within an assembly language program, I have used Basic to check it out and then hand-compiled just that small portion into a larger program. It works.

Rush Caley, LIVE!

Like a great many people in this country, much of my attention has been given to the war in the Persian Gulf. One of the fascinating points of interest to me is the insistence on the part of the press to ask stupid questions at Pentagon briefings and other such gatherings. Following are examples of such questions.

1. How long will the war last?
2. When will the ground war begin?
3. Will Sadaam Hussein use chemical and biological weapons?

The list could go on and on; but you see the pattern. All of this led me to thinking about some of the unanswered questions that nag me sometimes late into the night.

But before I go further, I should explain that I am a person given to listmaking. For some reason it helps me to compartmentalize things into nice little packages I can open from time to time and study. There are lists common to most of us that we live by. We make grocery lists, lists of "things to do", address lists, Christmas lists, and so on. But me? I go a step further. I have a list of pet peeves, a list of movies I have on tape, a list of historical people I most admire. It goes on ad nauseam ad infinitum.

That brings me back to this particularly favorite list of mine - a list of questions to which I really need the answers. Now I realize that I may not get the answers to many of these while still on this side of Paradise; but when I get to the other side, here's a few things I'm going to ask about straightaway:

1. Was Bruno Hauptmann innocent of killing the Lindberg baby?
2. Were the Kennedy brothers truly victims of lone assassins? What was the true nature of the conspiracy surrounding the death of President Kennedy?
3. What was the real cause of the extinction of the dinosaurs?
4. How does the bird in a cuckoo clock know when to come out?
5. Is labor racketeer James Hoffa really buried in a baseball stadium?

6. Did William Shakespeare write all of his plays?
7. Was the moon landing in July of 1969 real? Or was it, as some say, staged and filmed in Colorado?
8. Was there really a King Arthur; and more importantly, did Merlin the magician really live his life backwards?
9. The world spins on its axis at over 25,000 MPH. If everyone in the world could theoretically jump up in the air at precisely the same moment, would the world spin out from underneath us?
10 We know that time is an artificial measurement created by human beings. If time does not truly exist, why are we trapped within a specific portion of it?

Anyway, you might want to try making your own list. It is very relaxing and takes one's mind away from ROMS, RAMS, nanoseconds, CRT glare, and other such electronic worries. It can provide excellent mental exercise; but most of all it's fun!
RTC

- NEXT MONTH -

The last part of Bob van der Poel's informative "Selections in 'C' will continue with solutions to even more complex "C" programming stumbling blocks. Ron Anderson's "Beginner's Corner" series will continue with its adventures in assembly language programming the 68XXX processor. I am confident Rush Caley will come up with a subject of special interest to all of us.

In addition to the above, there will be a surprise for all of us.

Of course I'd like to see a fist full of ads in the new Classified Ads column, and enough time should have elapsed for us to get enough letters for a "Letters" column. All in all, it will be an issue of wanted information making for high interest. I'm looking forward to it and hearing from you.

Beginner's Corner

By Ron Anderson

Now we are about to get into something that is most useful. Let's write a "Filter" program. Basically a filter program is one that reads an input file one character at a time, performs some conversion on the text and writes the altered text to an output file. In simplest form, perhaps, it does something trivial. We could start with a program to reduce all multiple spaces in a file to single ones, or even more basic, one that converts all lower case characters in a file to upper case. Of course if we can do that, we can do the reverse.

```
*
* UPPER CASE UTILITY FOR SK*DOS
  /68K
* A PROTOTYPE "FILTER" PROGRAM
*
* THIS ONE CONVERTS LOWER CASE
  LETTERS TO UPPER
* FROM FILE TO FILE
*
* SYNTAX:  UPCASE INFILENAME
  OUTFILENAME
*
* INFILE MUST EXIST, OUTFILE
  MUST NOT
* CONVERTS ONLY a-z to A-Z.  ALL
  OTHER CHARACTERS
* NOT CHANGED.
*
* EQUATES TO SK*DOS
*
FCBERR EQU 1
VPOINT EQU $A000
DEFEXT EQU $A024
PSTRNG EQU $A035
FCLOSE EQU $A008
FOPENR EQU $A005
FOPENW EQU $A006
FREAD EQU $A001
FWRITE EQU $A002
GETNAM EQU $A023
PCRLF EQU $A034
PERROR EQU $A037
PUTCH EQU $A033
WARMST EQU $A01E
*
UPPER BRA.S START GOTO START
VER DC.W $0100 VERSION NUMBER
START DC VPOINT
 MOVE.L A6,A0 SAVE USER FCB
              POINTER FOR OUTPUT
 LEA INFCB(PC),A3 POINTER TO
```

```
                            INPUT FCB
 MOVE.L A3,A4 POINTER
 DC GETNAM GET FILE SPEC
 BCS HELP
 MOVE.B #1,D4 DEFAULT EXTENSION
 DC DEFEXT
 MOVE.L A0,A4 OUTPUT FCB POINTER
 DC GETNAM
 BCS HELP
 MOVE.B #1,D4 DEFAULT EXTENSION TXT
 DC DEFEXT DEFAULT EXTENSION
*
* NOW OPEN THE FILES
 MOVE.L A3,A4 INFILE POINTER
 DC FOPENR
 BNE.S ERROR IF NOT ZERO
 MOVE.L A0,A4 OUTFILE POINTER
 DC FOPENW OPEN FOR WRITE
 BNE.S ERROR IF NOT ZERO
*
* MAIN LOOP TO READ AND WRITE EACH CHAR
*
MAIN MOVE.L A3,A4 POINT TO INFILE
 DC FREAD GO READ NEXT CHAR
 BNE.S ERROR
*
* HERE IS THE FILTER THAT COMPARES CHAR WITH
* a-z and changes to A-Z
* This section could be replaced with code to do
* other functions. The rest of the pgm only deals
* with opening and closing the files etc.
*
 CMP.B #'a',D5
 BLT.S CHAR1 ASCII VALUE TOO LOW TO BE IN RANGE
 CMP.B #'z',D5
 BGT.S CHAR1 ASCII VALUE TOO HIGH TO BE IN RANGE
 SUB.B #$20,D5 CHANGE IT FROM LOWER TO UPPER
*
* END OF FILTER, WRITE IT TO OUTPUT FILE
*
CHAR1 MOVE.L A0,A4 OUTPUT FILE FCB POINTER
 MOVE.B D5,D4 CHAR READ INTO D5, WRITTEN FROM D4
 DC FWRITE WRITE TO OUTPUT FILE
 BRA.S MAIN AND CONTINUE
*
* ERROR HANDLER
*
ERROR CMP.B #8,FCBERR(A4)
 BEQ.S EXIT
 DC PERROR PRINT ERROR CODE
EXIT BSR.S CLOSE CLOSE THE FILE
 DC WARMST RETURN TO SKDOS
*
* CLOSE SUBROUTINE
*
CLOSE MOVE.L A0,A4 POINT TO OUTPUT FILE FCB
 DC FCLOSE CLOSE FILE
 MOVE.L A3,A4 POINTER TO INPU FILE FCB
 DC FCLOSE
 RTS
*
HELP LEA HLPMSG(PC),A4
 DC PSTRNG
 DC WARMST
INFCB DS.B 608
HLPMSG DC.B "Syntax: UPCASE INFILENAME
             OUTFILENAME",$0D,$0A
 DC.B "UPCASE reads an existing file converting all
       the lower",$0D,$0A
 DC.B "case letters from infile to upper case and
       writing the",$0D,$0A
 DC.B "result to outfile. Default extensions .TXT
       for both files",$0D,$0A
 DC.B "and both default to the work drive.",
       $0D,$0A,$04
 END UPPER
```

There are several things to discuss. First, if GETNAM has an error, that means that a proper file specification (or rather two of them) are not present. In that case, the HELP message is to be printed for the user. Try running UPCASE without any filenames and you will get the help message.

We've used SK*DOS' system file control block for the output file FCB rather than creating our own. The choice is ours and it really doesn't matter much. Most writers of utilities do take advantage of the system FCB which is used to load the program, but then is available to the user program.

The FILTER action takes place in five lines clearly marked in the program listing. Suppose we wanted to eliminate multiple spaces in a text file. We could simply replace a section of our code with the following:

This line is added just before the main loop

```
 CLR.B D0 USE FOR MULTIPLE SPACE FLAG
*
* MAIN LOOP TO READ AND WRITE EACH CHAR
*
MAIN MOVE.L A3,A4 POINT TO INFILE
 DC FREAD GO READ NEXT CHAR
 BNE.S ERROR
*
* This code goes in the Filter slot
* It will remove multiple spaces in a file
 CMP.B #$20,D5
 BNE.S NOTSP
 TST.B D0
 BNE.S MAIN IF WE'VE OUTPUT A SPACE SKIP THIS ONE
 MOVE.B #$FF,D0 SET SPACE FLAG AFTER OUTPUT OF ONE
 BRA.S CHAR1
NOTSP CLR.B D0 NOT A SPACE SO CLEAR SPACE FLAG
*
* END OF FILTER, NOW WRITE IT TO OUTPUT FILE
*
```

Granted this is a little more complex than simply making lower case characters upper case ones, but it is not terribly difficult. You can perhaps see the usefulness of such a program. Suppose for example, that we have somehow imported an MS-DOS text file that uses $0A to terminate a line. It would be simple to change $0A to $0D wherever it is found. That is an easy filter. Suppose you had written a book and when done, wanted to change the name of a character in the book from Frederick to Paul. That would be a bit harder, but you could still write a filter program to make the change.

Well, by now you should be pretty much comfortable with assembler, though we have not touched on a number of the instructions of the 68000. There are the MUL and DIV instructions, the DBcc set, of which it only makes sense to use DBEQ or possibly DBMI. These instructions are useful for repeating a loop a predetermined number of times, and they save a small amount of code. The MULU and MULS (multiply unsigned and multiply signed) instructions are described in the user manual, but unless you are at least somewhat familiar with binary arithmetic, they won't be easy to understand immediately. The same can be said of the DIV instructions.

- Style in Assembler Programs -

So far we have avoided a discussion of Style in writing assembler programs. None of the "unassembled" source listings presented so far have been tabbed. That is, the labels and comments started in the first column, the operation mnemonics in the second or one space after a label, etc. The assembler will tabularize the listing when it runs, and you can have the assembler prepare a listing for you to study. Some people like to tab their source code so it is easier to read. I have no objection to that. My non-tabbing is probably a holdover from when a floppy held 90K bytes, and I had 32K of RAM to work with. It was advantageous to keep source files short.

I am going to borrow a short section of code from Marion Systems drivers for their SCSI interface for the hard disk. I'm sure Tom Oberheim won't mind terribly. What the code does is relatively unimportant, but it transmits a 6 byte pre-composed message to the SCSI as a command. Let's first do a completely stripped down version of it:

```
PUT_COM_6 MOVE.B #TCRCDM,SCSITCR
PC6COMWAIT BTST.B #BSRPHMB,SCSIBSR
 BEQ.L PC6COMWAIT
 MOVEQ #5,D1
PC6LOOP MOVE.B (A0)+,D0
 BSR.L PUT_BYTE
 DBF D1,PC6LOOP
 MOVEQ #0,D7
 RTS
```

That is a total of 9 lines of code. The author used long labels and symbols hoping to make the code more readable. I think it is, but only to someone already familiar with the SCSI device. For example, SCSIBSR is the SCSI Buss Status Register, etc. An effort was made to make the symbols mnemonic and suggestive of the thing they represented. A step in making this more readable would be to tab it.

```
PUT_COM_6    MOVE.B #TCRCDM,SCSITCR
PC6COMWAIT   BTST.B #BSRPHMB,SCSIBSR
             BEQ.L PC6COMWAIT
             MOVEQ #5,D1
PC6LOOP      MOVE.B (A0)+,D0
             BSR.L PUT_BYTE
             DBF D1,PC6LOOP
             MOVEQ #0,D7
             RTS
```

The next step would be to comment the lines of code:

```
PUT_COM_6    MOVE.B #TCRCDM,SCSITCR Only CD should
                  be asserted
PC6COMWAIT   BTST.B #BSRPHMB,SCSIBSR Test the phase
                  match bit
             BEQ.L PC6COMWAIT Wait for this phase
             MOVEQ #5,D1 Will send 6 bytes
PC6LOOP      MOVE.B (A0)+,D0 Get byte to send
             BSR.L PUT_BYTE Send it
             DBF D1,PC6LOOP Loop until all bytes
                  sent
             MOVEQ #0,D7 Zero D7 to signal
                OK
             RTS
```

So far I am with this. The author, however, made it into the following:

```
***********
*    PUT_COM_6:   ISSUE A 6-BYTE SCSI COMMAND
*
*    PUT_COM_6 First checks that the target is
                ready to receive
*    a command.  It then sends 6 command bytes.
*
*    Entry:
*     Ø points to bytes to send
*
*    Uses:
*          DØ for byte transfer
*          D1 for loop index
*
*    Returns:
*          D7 = Ø and Z flag set only if all is
           well.
************
PUT_COM_6
*  Set Target Command Register to COMMAND phase to
   check for
*  mismatch in lines.
*
   MOVE.B      #TCRCDM,SCSITCR Only Cd should be
               asserted
*  Wait for command phase
 PC6COMWAIT
   BTST.B      #BSRPHMB,SCSIBSR Test the phase match
               bit
   BEQ.L       PC6COMWAIT Wait for this phase
 *  Send 6 bytes
   MOVEQ        #5,D1 Will send 6 bytes
PC6LOOP
   MOVE.B      (AØ)+,DØ Get byte to send
   BSR.L       PUT_BYTE Send it
   DBF         D1,PC6LOOP Loop until all bytes sent
   MOVEQ       #Ø,D7 Zero D7 to signal OK
PC6EXIT
*  All bytes sent
   RTS
```

If I count correctly that is 49 lines of program for 9 lines of actual code. First note that ASM allows labels on lines by themselves. They associate with the next line of code. PUT_COM_6 (the label) is 6 lines away from the first line of code. The label PC6EXIT does no harm, but there are no references to it. If you really feel that commenting to this degree HELPS you to understand the program (or more to the point, helps someone else to understand it), then so be it. To my mind, however, the code gets lost in the comments. The section of program of which this is part, is a 14 page listing with two pages of actual code. In my opinion it would be easier to understand and use if it were four or five pages. In a more extreme example I could have chosen from the same listing of SCSI routines, there is one called WAIT_AWHILE. It uses 26 lines to document TWO LINES of assembler code.

My rules for assembler code commenting are:

1. Each routine or major section should have a heading describing what the routine does, what is passed to the routine in which registers, what is returned, and which registers the routine uses but does not restore. The latter is particularly important with the 68ØØØ since it has so many registers.

2. A comment is not needed for EVERY line. After you've written two assembler programs an instruction like DBEQ D1,LOOP doesn't need an informationless comment like "Go Around Again" or worse, "Bump The Counter". An instruction like TST.B D1 doesn't need the comment "Set the Flags". Comments should add information to what is already there, not just spell out the instruction.

3. A blank line can be used to separate minor sub-parts of routines without requiring a three line comment.

4. If a listing is tabbed, labels stick out like sore thumbs and don't need a line all their own.

If you don't like my rules,use your own, but be consistent. I would treat the above listing as follows:

```
************
*    PUT_COM_6 First checks that the target is
     ready to receive
*    a command. It then sends 6 command bytes.
*    Entry: AØ points to bytes to send
*    Returns: D7 = Ø and Z flag set if all is well
*    Uses: DØ for byte transfer
*          D1 for loop index
************
*  Set Target Command Register to COMMAND phase to
   check for
*  mismatch in lines.

PUT_COM_6 MOVE.B #TCRCDM,SCSITCR Only Cd should be
                asserted
*  Wait for command phase
*
PC6COMWAIT BTST.B
#BSRPHMB,SCSIBSR Test the phase match bit
 BEQ.L PC6COMWAIT Wait for this phase
*
*  Send 6 bytes
*
 MOVEQ #5,D1 Will send 6 bytes
PC6LOOP MOVE.B (AØ)+,DØ Get byte to send
 BSR.L PUT_BYTE Send it
 DBF D1,PC6LOOP Loop until all bytes sent
 MOVEQ #Ø,D7 Zero D7 to signal OK
 RTS
```

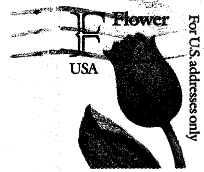
First, I would depend on the assembler to tab the listing later. Secondly, I would put all labels on the line to which they refer. Thirdly, things like RTS (ReTurn from Subroutine) hardly need a comment to tell us that we've reached the end of the code. Nor do they need an unused label to do the same.

When my son or my daughter had a paper to write in one of their classes and they wanted to use the computer as a word processor, they always wanted to set the printer to double space and 1Ø character per inch, as opposed to a nice proportional spacing printing I have that uses about 15 characters per inch. Why? They simply wanted the paper to look longer. Two pages are always more impressive than 2/3 page, right?

Unless you are a programming consultant and getting paid by the page, there is no need to make listings extra long. That wastes paper and wears out printers! Impress people with how short and simple your code is, not with how many pounds of paper it takes to print it.

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

WILMINGTON DE
PM
MAR 2
1991
19850



– Address Correction Requested –

– First Class –