68 NEWS

Issue No. 9
April 1990

IN THIS ISSUE

READ THIS!

Will YOU get the next issue of 68 News?

Maybe YES ... and maybe NO.

The code number above your name on the mailing label shows the number of the LAST issue of the 68 News you are scheduled to get. For example, if the code above your name says N9, then this issue – issue number 9 – is the last you will get.

There are three ways of extending your subscription:

(1) Subscribe for $10.

(2) Advertise. A half-page advertisement costs $10, and advertisers get a copy of the News.

(3) Send us an article. While we do not have much space, we will be happy to publish your article as space allows, and will send you a free subscription as well.

So don't forget -- if it says N9 above your name, then this is IT!

The *68 NEWS* is published and copyright © 1990 by Star-K Software Systems Corp, P. O. Box 209, Mt. Kisco NY, 10549. The editor is Peter A. Stark.

The subscription price is $10 per year. We accept display advertising at the rate of $10 per half-page (3.5" high by 4.5" wide). Readers are invited to contribute articles, letters, programs, tutorials, and other material for publication. We publish only material and advertising which, in the opinion of the editor, (a) applies to hardware or software for 68xx(x) type processors, and (b) is of a nature which would not normally be of interest to the major computer magazines. We simply do not have room for items of a very general nature, or items which pertain to very popular systems like the Macintosh or Amiga.

Please send articles or other material to us at the above address (preferably on disk); you may also fax us at (914) 241-8607, send it via modem to our BBS at (914) 241-3307, or call us at (914) 241-0287.

We thank you for your support.

The Solid-State Disk with Battery-Backup

by Dr. Mike Randall, PO Box 1320, WELLINGTON NEW ZEALAND, FAX 0064 4 710 977

It has always seemed to me a good idea to share the lessons learned the hard way developing software and hardware. After all, I have gained much from the work other people have shared with me. Pressure of work makes it easy to put off such altruism, however if the interest is there I will try to keep up a series of articles on such topics as the implementation of print spooling using a device driver, my MODULA-2 compiler for SK*DOS, and my dual-processor bussed computer which runs SK*DOS on both 68000 and 6809 communicating via dual-ported RAM.

This article describes the installation, formatting and driver software for XTRD, a RAM-DISK board that occupies the PC bus on my PT-2 computer. It has all the speed advantage of the software RAMDISK but leaves the system RAM free for other uses - and it is battery- backed up, so it retains its data even when the main computer is turned off!

I will not go into details of the hardware implementation; anyone interested can contact me directly. Sufficient information for my present purposes is displayed in the block diagram of XTRD on the next page. The memory array is accessed through a 256 byte window (odd bytes) in the PC bus area at addresses $DDFC01 through $DDFDFF. Two latches select the particular page (or sector) currently available for access. These latches can be thought of as selecting the current physical track and sector of the RAM-DISK.

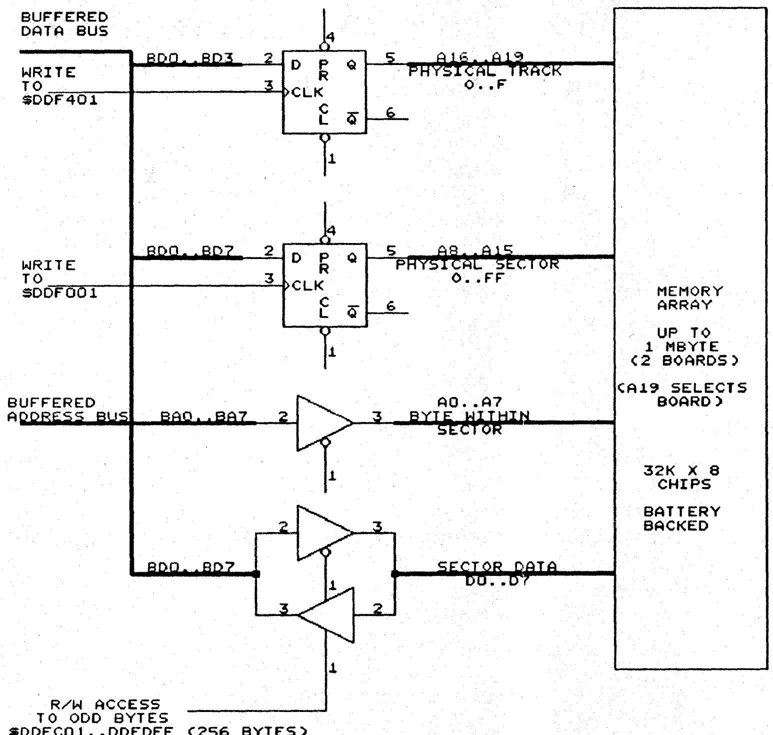
*(Editor's note: I've taken the liberty of modifying Mike's block diagram a bit, so let me explain. The memory array at the right gets the 20 address lines called A19 through A0. Of these, A16 through A19 select what SK*DOS thinks is a track between 0 and 15; these lines are latched in a group of flip-flops (only one is shown) whenever the program writes to address $DDF401 (not shown is the address decoder which decodes this address). In Mike's case, the memory array is mounted on two boards, each with 1/2 megabyte, and A19 actually selects which of the two boards is used – tracks 0 through 7 are on one board, 8 through 15 on the other.*

*Similarly, address lines A15 through A8 are latched in a group of flip-flops which appear at address $DDF001, and these select what SK*DOS thinks is a sector between 0 and 255. The track and sector numbers are latched because the disk driver writes these before transferring actual data.*

Memory address lines A7 through A0 come from the address bus on the expansion connectors through buffers, and select the actual byte within the sector. Likewise, the eight data lines D7 through D0 also come from the bus, but these go through bidirectional transceivers (again, only one is shown) since this data has to go in one direction for reading, and the other for writing. These buffers appear at the 256 odd addresses starting at $DDFC01.)

The program XTRD.COM serves to install the memory resident driver code (if it is not already installed), to assess how much XTRD memory is available and to format the XTRD memory (if it is not already formatted).

SK*DOS provides hooks for secondary disk I/O routines - see chapter 13 of the Users' Manual. Up to 3 sets of secondary routines may be installed, each set



Block diagram of the XTRD solid state disk.

needs a sector read routine, a sector write routine and a disk ready check routine. The space set aside for these hooks is at SECTRD, SECTWR and SECCHK. When no secondary routines are installed the space is filled by RTS codes.

By convention, the high order bits of FCBPHY (the physical drive number byte in the FCB) are used to indicate drive type. I use bit 6 which indicates type O (Other) for XTRD. Thus XTRD can co-exist with the standard RAMDISK (type R). I make no assumption as to previously installed secondary drivers except that none is of type O. XTRD.COM uses its CHKSEC routine to find the next available slot for its routines. The program uses the SK*DOS flag space at SECFL3 to indicate successful installation of its drivers.

The driver routines are made memory resident by setting OFFINI to a new value just past the resident code. The following transient code is freed once installation is complete.

I have chosen to use a logical track/sector sequence with sectors numbered from 1 to $20 (so it is not necessary to use the +T option with COPY). These must be converted to physical track/sec before writing to the latches. Movement of data between the odd bytes of the XTRD memory window and the FCB uses the MOVEP.L command for efficiency.

The formatting technique is fairly standard. An image of the system information sector (SIS) is set up in the transient code area and the details filled in there before it is eventually copied to its place in XTRD. The program first checks to see if XTRD is already formatted. The word $AA55 at the very beginning of XTRD memory (physical track/sec $0000) shows this. The user is given the opportunity to reformat if desired. Before formatting, XTRD memory is tested for the presence of each successive memory chip. (So you don't have to completely populate the board). Thus the sector address of the last free sector and the number of free sectors can be calculated.

The initial directory structure is then set up, with links from logical track/sec $0005 to $0020, and the directory data cleared. Following this the free chain links are set up. Finally the RAM copy of the SIS is set up and copied to logical track/sec $0003, and the 'formatted' flag is written.

Installation is the last step of the process. Space is found in the secondary disk I/O routine areas and the code JSR.L xxxx set in for each of the routines SECRD2, SECWR2 and SECCHK. The subroutine SETXRD is called to adjust the SK*DOS DRUSED table. The next free logical drive is assigned to XTRD and MAXDRV updated, and the user is informed. Finally OFFINI is set to the end of the resident code and the flag $AA55 along with its address stored at SECFL3. These give a reasonably secure check of installation should XTRD.COM be called again. (You might want to reformat but can't be allowed to install another set of drivers).

I hope this has been instructive. If anyone is interested in the XTRD board they should contact me directly. Other PC ramdisk boards might be suitable for the PT-2 and my software could be modified to allow for their hardware interface. Mine works and really improves productivity. Frequently used programs that normally take an age to load can be copied to XTRD which can be used as the work drive. The compile edit recompile sequence is very much faster. The battery backed RAM holds data for several days so backing up work files is not exhausting.

For those of you who would like to adapt this idea to different hardware, the actual code for XTRD.COM follows:

```
                 *=======================================
                 * VERSION TO USE SECTORS 1..$20
                 *=======================================

                 * XTRD EQUATES
0000 00DDF401 RDTRK     EQU      $DDF401
0000 00DDF001 RDSEC     EQU      $DDF001
0000 00DDFC01 DSKIOB    EQU      $DDFC01
                        LIB      SKEQUATE.TXT
                 * SK*DOS / 68K EQUATES FOR USER PROGRAMS

                 *=======================================
                 * RESIDENT CODE
                 *=======================================
0000 082C0006. SECRD2   BTST     #6,FCBPHY(A4)
0006 6602               BNE.S    SEMRD drive type 0(ther)
0008 4E75               RTS
000A 48E70306 SEMRD     MOVEM.L  D6-D7/A5-A6,-(A7)
                 * convert logical trk/sec to physical
000E 2C3C0000.          MOVE.L   #0,D6
0014 1C2C0022           MOVE.B   FCBCTR(A4),D6
0018 EB46               ASL.W    #5,D6
```

```
001A DC2C0023            ADD.B    FCBCSE(A4),D6
001E 6604                BNE.S    RDRO
0020 06460100            ADD.W    #$100,D6
0024 13C600DD.RDRO       MOVE.B   D6,RDSEC
002A E046                ASR.W    #8,D6
002C 13C600DD.           MOVE.B   D6,RDTRK
                  * move data to FCB
0032 4BEC0060            LEA      FCBDAT(A4),A5
0036 2C3C0000.           MOVE.L   #63,D6 256-BYTE SECTORS
003C 2C7C00DD.           MOVE.L   #DSKIOB,A6
0042 0F4E0000 RDR1       MOVEP.L  0(A6),D7
0046 DDFC0000.           ADD.L    #8,A6
004C 2AC7                MOVE.L   D7,(A5)+
004E 51CEFFF2            DBRA     D6,RDR1
0052 4CDF60C0            MOVEM.L  (A7)+,D6-D7/A5-A6
0056 DFFC0000.           ADD.L    #8,A7 RETURN TO PRIMARY CALLER
005C 003C0004            OR.B     #4,CCR SET Z
0060 4E75                RTS

0062 082C0006.SECWR2     BTST     #6,FCBPHY(A4)
0068 6602                BNE.S    SEMWR drive type 0(ther)
006A 4E75                RTS
006C 48E70306 SEMWR      MOVEM.L  D6-D7/A5-A6,-(A7)
                  * convert logical trk/sec to physical
0070 2C3C0000.           MOVE.L   #0,D6
0076 1C2C0022            MOVE.B   FCBCTR(A4),D6
007A EB46                ASL.W    #5,D6
007C DC2C0023            ADD.B    FCBCSE(A4),D6
0080 6604                BNE.S    RDWO
```

```
0082 06460100           ADD.W    #$100,D6
0086 13C600DD. RDW0     MOVE.B   D6,RDSEC
008C E046               ASR.W    #8,D6
008E 13C600DD.          MOVE.B   D6,RDTRK
               * MOVE DATA FROM FCB
0094 4BEC0060           LEA      FCBDAT(A4),A5
0098 2C7C00DD.          MOVE.L   #DSKIOB,A6
009E 2C3C0000.          MOVE.L   #63,D6 256-BYTE SECTORS
00A4 2E1D      RDW1     MOVE.L   (A5)+,D7
00A6 0FCE0000           MOVEP.L  D7,0(A6)
00AA DDFC0000.          ADD.L    #8,A6
00B0 51CEFFF2           DBRA     D6,RDW1
00B4 4CDF60C0           MOVEM.L  (A7)+,D6-D7/A5-A6
00B8 DFFC0000.          ADD.L    #8,A7 RETURN TO PRIMARY CALLER
00BE 003C0004           OR.B     #4,CCR SET Z
00C2 4E75               RTS

00C4 082C0006. secchk   BTST     #6,FCBPHY(A4)
00CA 6602               BNE.S    SEMCHK
00CC 4E75               RTS
00CE DFFC0000. SEMCHK   ADD.L    #8,A7 RETURN TO PRIMARY CALLER
00D4 003C0004           OR.B     #4,CCR SET Z
00D8 4E75               RTS

00DA AA55      RESFLG   DC.W     $AA55
00DC 000000DC  TRANST   EQU      *

               *========================================
               * TRANSIENT CODE
               *========================================
00DC 000000DD  SISDAT   EQU      *+1
00DC 00000000.          DC.W     0,0,0,0,0,0,0,0
00EC 00                 DC.B     0
00ED 58545F52.          DC.B     'XT_RAM_DISK'
00F8 0000               DC.W     0
00FA 0101      FSFREE   DC.W     $101
00FC 0000      LSFREE   DC.W     0 fill in later
00FE 0000      NRFREE   DC.W     0 fill in later
0100 0000      MTHCR    DC.W     0 fill in later
0102 00000101  DATCR    EQU      MTHCR+1
0102 0000      YRCR     DC.W     0 fill in later
0104 00000103  MAXTRK   EQU      YRCR+1
0104 2000      MAXSEC   DC.B     $20,0
                        RPT      14
0106 00000000.          DC.W     0,0,0,0,0,0,0,0
0116 00000000.          DC.W     0,0,0,0,0,0,0,0
0126 00000000.          DC.W     0,0,0,0,0,0,0,0
0136 00000000.          DC.W     0,0,0,0,0,0,0,0
0146 00000000.          DC.W     0,0,0,0,0,0,0,0
0156 00000000.          DC.W     0,0,0,0,0,0,0,0
0166 00000000.          DC.W     0,0,0,0,0,0,0,0
0176 00000000.          DC.W     0,0,0,0,0,0,0,0
0186 00000000.          DC.W     0,0,0,0,0,0,0,0
0196 00000000.          DC.W     0,0,0,0,0,0,0,0
01A6 00000000.          DC.W     0,0,0,0,0,0,0,0
01B6 00000000.          DC.W     0,0,0,0,0,0,0,0
01C6 00000000.          DC.W     0,0,0,0,0,0,0,0
01D6 00000000.          DC.W     0,0,0,0,0,0,0,0

               * POINT TO TRK/SEC
               * ENTRY:- D0.W = TRK/SEC (PHYSICAL)
               * USES D1
```

```
01E6 3200     POINT    MOVE.W   D0,D1
01E8 13C100DD.         MOVE.B   D1,RDSEC
01EE E041              ASR.W    #8,D1
01F0 13C100DD.         MOVE.B   D1,RDTRK
01F6 4E75              RTS

              * SET UP A LINK
              * ENTRY:- D0.W = TRK/SEC (PHYSICAL)
              *         D2.W = LINK TRK/SEC(LOGICAL)
              *         A2   = DSKIOB
              * USES D1 (POINT)

01F8 6100FFEC LINK     BSR      POINT
01FC 058A0000          MOVEP.W  D2,0(A2)
0200 4E75              RTS

              * MESSAGES
0202 58545244.AVAIL    DC.B     'XTRD total sectors = ',4
0218 416C7265.ASK      DC.B     'Already formatted - Reformat? ',4
0237 52657369.IN       DC.B     'Resident code already installed',4
0257 52657369.INST     DC.B     'Resident code installed',4
026F 4E6F2072.NOROOM   DC.B     'No room in secondary table',4
028A 4E6F2058.NORAM    DC.B     'No XTRD RAM present',4
029E 58545244.DRV      DC.B     'XTRD is logical drive ',4

02B6 A000     START    DC       VPOINT
02B8 206E0346          MOVE.L   DOSORG(A6),A0 A0=DOSORG THROUGHOUT
              * CHECK IF ALREADY FORMATTED
02BC 303C0000          MOVE.W   #0,D0
02C0 6100FF24          BSR      POINT
02C4 247C00DD.         MOVE.L   #DSKIOB,A2
02CA 010A0000          MOVEP.W  0(A2),D0
02CE 0C40AA55          CMP.W    #$AA55,D0
02D2 6614              BNE.S    FORMAT
02D4 49FAFF42          LEA      ASK(PC),A4
02D8 A035              DC       PSTRNG
02DA A029              DC       GETCH
02DC 0205005F          AND.B    #$5F,D5
02E0 0C050059          CMP.B    #$59,D5
02E4 66000156          BNE      INSTAL

              * ASSESS XTRD MEMORY (A2 STILL = DSKIOB)
02E8 303C0000 FORMAT   MOVE.W   #0,D0
02EC 6100FEF8 AM1      BSR      POINT
02F0 070A0000          MOVEP.W  0(A2),D3 READ XTRD MEMORY
02F4 343CA55A          MOVE.W   #$A55A,D2
02F8 058A0000          MOVEP.W  D2,0(A2) WRITE PATTERN
02FC 050A0000          MOVEP.W  0(A2),D2 READ BACK
0300 0C42A55A          CMP.W    #$A55A,D2 COMPARE
0304 660E              BNE.S    AM2 NO RAM CHIP THERE
0306 078A0000          MOVEP.W  D3,0(A2) RESTORE DATA
030A 06400080          ADD.W    #$80,D0 NEXT CHIP
030E 0C401000          CMP.W    #$1000,D0
0312 66D8              BNE.S    AM1 TRY NEXT CHIP
0314 4A40     AM2      TST.W    D0
0316 672C              BEQ.S    AM3
0318 49FAFEE8          LEA      AVAIL(PC),A4
031C A035              DC       PSTRNG
031E 283C0000.         MOVE.L   #0,D4
0324 2A3C0000.         MOVE.L   #0,D5
032A 3800              MOVE.W   D0,D4
032C A038              DC       OUT5D REPORT TO CALLER
032E 04400001          SUB.W    #1,D0
```

```
                    * DO IS NOW PHYSICAL ADDR OF LSFREE
0332 45FAFDC8               LEA       LSFREE(PC),A2
0336 3480                   MOVE.W    DO,(A2) CHANGE TO LOGICAL LATER
0338 04400020               SUB.W     #$20,DO
033C 45FAFDCO               LEA       NRFREE(PC),A2
0340 3480                   MOVE.W    DO,(A2)
0342 6008                   BRA.S     DIR

                    *  WHAT IF NO RAM?
0344 49FAFF44 AM3           LEA       NORAM(PC),A4
0348 A035                   DC        PSTRNG
034A A01E                   DC        WARMST

                    * DIR LINKS
034C 247C00DD.DIR           MOVE.L    #DSKIOB,A2
0352 303C0005               MOVE.W    #5,DO
0356 343C0006               MOVE.W    #6,D2
035A 6100FE9C DR1           BSR       LINK
035E 06400001               ADD.W     #1,DO
0362 06420001               ADD.W     #1,D2
0366 0C400020               CMP.W     #$20,DO
036A 66EE                   BNE.S     DR1
036C 243C0000.              MOVE.L    #0,D2
0372 6100FE84               BSR       LINK
                    * CLEAR DIR DATA
0376 303C0005               MOVE.W    #5,DO
037A 6100FE6A CD1           BSR       POINT
037E 303C0000               MOVE.W    #0,DO
0382 323C007E               MOVE.W    #126,D1
0386 247C00DD.              MOVE.L    #DSKIOB+4,A2
038C 018A0000 CD2           MOVEP.W   DO,0(A2)
0390 D5FC0000.              ADD.L     #4,A2
0396 51C9FFF4               DBRA      D1,CD2
039A 247C00DD.              MOVE.L    #DSKIOB,A2
03A0 010A0000               MOVEP.W   0(A2),DO
03A4 4A40                   TST.W     DO
03A6 66D2                   BNE.S     CD1

                    * FREE CHAIN LINKS
03A8 247C00DD.              MOVE.L    #DSKIOB,A2
03AE 303C0021               MOVE.W    #$21,DO
03B2 343C0102               MOVE.W    #$102,D2
03B6 6100FE40 F1            BSR       LINK
03BA 06400001               ADD.W     #1,DO
03BE 06420001               ADD.W     #1,D2
03C2 0C020021               CMP.B     #$21,D2
03C6 6604                   BNE.S     F2
03C8 064200E0               ADD.W     #$E0,D2
03CC B07AFD2E F2            CMP.W     LSFREE(PC),DO
03D0 66E4                   BNE.S     F1
03D2 343C0000               MOVE.W    #0,D2
03D6 6100FE20               BSR       LINK

                    * SIS - CORRECT LSFREE TO LOGICAL
03DA 45FAFD20               LEA       LSFREE(PC),A2
03DE 3012                   MOVE.W    (A2),DO
03E0 02400FE0               AND.W     #$FE0,DO
03E4 E740                   ASL.W     #3,DO
03E6 0000001F               OR.B      #$1F,DO
03EA 3480                   MOVE.W    DO,(A2)
                    * SET MAXTRK
03EC E040                   ASR       #8,DO
03EE 45FAFD13               LEA       MAXTRK(PC),A2
```

```
03F2 1480              MOVE.B   D0,(A2)

03F4 45FAFD0A          LEA      MTHCR(PC),A2
03F8 14EE02EE          MOVE.B   CMONTH(A6),(A2)+
03FC 14EE02EF          MOVE.B   CDAY(A6),(A2)+
0400 14AE02F0          MOVE.B   CYEAR(A6),(A2)
0404 303C0003          MOVE.W   #3,D0
0408 6100FDDC          BSR      POINT
040C 247C00DD.         MOVE.L   #DSKIOB,A2
0412 47FAFCC9          LEA      SISDAT(PC),A3
0416 303C00FF          MOVE.W   #255,D0
041A 149B      S1      MOVE.B   (A3)+,(A2)
041C D5FC0000.         ADD.L    #2,A2
0422 51C8FFF6          DBRA     D0,S1

               * SET XTRD FORMAT FLAG
0426 303C0000          MOVE.W   #0,D0
042A 6100FDBA          BSR      POINT
042E 247C00DD.         MOVE.L   #DSKIOB,A2
0434 323CAA55          MOVE.W   #$AA55,D1
0438 038A0000          MOVEP.W  D1,0(A2)

               * CHECK IF ALREADY INSTALLED
043C 0C68AA55.INSTAL   CMP.W    #$AA55,$90(A0) SECFL3 has $AA55 if in-
                                 stalled
0442 6612              BNE.S    I1
0444 22680092          MOVE.L   $92(A0),A1 = RESFLG in resident code
0448 0C51AA55          CMP.W    #$AA55,(A1) is code still there?
044C 6608              BNE.S    I1
               * ALREADY INSTALLED
044E 49FAFDE7          LEA      IN(PC),A4
0452 A035              DC       PSTRNG
0454 A01E              DC       WARMST

               * INSTALL
0456 43FAFBA8 I1       LEA      SECRD2(PC),A1
045A 45E80020          LEA      $20(A0),A2 beginning of SECREAD area
045E 6154              BSR.S    CHKSEC find free space
0460 664A              BNE.S    I2 NO ROOM
0462 34FC4EB9          MOVE.W   #$4EB9,(A2)+ JSR.L install JSR.L SECRD2
0466 2489              MOVE.L   A1,(A2) ADDRESS
0468 43FAFBF8          LEA      SECWR2(PC),A1
046C 45E80034          LEA      $34(A0),A2 beginning of SECWRITE area
0470 61000042          BSR      CHKSEC find free space
0474 6636              BNE.S    I2 NO ROOM
0476 34FC4EB9          MOVE.W   #$4EB9,(A2)+ JSR.L install JSR.L SECWR2
047A 2489              MOVE.L   A1,(A2) ADDRESS
047C 43FAFC46          LEA      secchk(PC),A1
0480 45E80070          LEA      $70(A0),A2 beginning of SECCHECK area
0484 612E              BSR.S    CHKSEC find free space
0486 6624              BNE.S    I2 NO ROOM
0488 34FC4EB9          MOVE.W   #$4EB9,(A2)+ JSR.L install JSR.L secchk
048C 2489              MOVE.L   A1,(A2) ADDRESS

048E 613A              BSR.S    SETXRD SET DRIVE PARAMS
0490 43FAFC4A          LEA      TRANST(PC),A1 end of resident code
0494 21490018          MOVE.L   A1,$18(A0) OFFINI set to it
0498 43E80090          LEA      $90(A0),A1 SECFL3
049C 45FAFC3C          LEA      RESFLG(PC),A2
04A0 32D2              MOVE.W   (A2),(A1)+ store flag $AA55 and its
                                   address there
04A2 228A              MOVE.L   A2,(A1) to show installed
```

```
04A4 49FAFDB1          LEA      INST(PC),A4 indicate installation
04A8 A035              DC       PSTRNG
04AA A01E              DC       WARMST

              * NO ROOM IN SECONDARY DRIVE ROUTINE TABLES
04AC 49FAFDC1 I2       LEA      NOROOM(PC),A4
04B0 A035              DC       PSTRNG
04B2 A01E              DC       WARMST

              * ROUTINE TO CHECK THERE IS SUFFICIENT ROOM IN SECONDARY
              * DISK ROUTINE TABLE
              * ENTRY:- A2 = START OF SECONDARY AREA
              * EXIT:-  A2 = NEXT ENTRY - Z SET IF ENOUGH ROOM FOR JSR.L
XXXX
              * USES D0
04B4 303C0007 CHKSEC   MOVE.W   #7,D0 NEED 3 WORDS
04B8 0C524E75 C1       CMP.W    #$4E75,(A2) RTS
04BC 670A              BEQ.S    C2 return with Z set when A2 = RTS code
04BE D5FC0000.         ADD.L    #2,A2 ADDA DOESN'T CHANGE CC
04C4 51C8FFF2          DBRA     D0,C1
              * NE holds here if loop falls thru
04C8 4E75     C2       RTS

              * ROUTINE TO FIND NEXT FREE LOGICAL DRIVE
              * SET AT XTRD ('00')
              * SET MAXDRV TO IT
              * ASSUMES ENOUGH SPACE
```

```
04CA 45E8013C SETXRD   LEA      $13C(A0),A2 = DRVUSED TABLE
04CE 4200              CLR.B    D0
04D0 0C1A0000 SX1      CMP.B    #0,(A2)+
04D4 67000008          BEQ      SX2 1st free entry in table
04D8 06000001          ADD.B    #1,D0
04DC 60F2              BRA.S    SX1
04DE 153C0040 SX2      MOVE.B   #$40,-(A2) Lx=00
04E2 1D400322          MOVE.B   D0,MAXDRV(A6) LD=MD
04E6 49FAFDB6          LEA      DRV(PC),A4
04EA A035              DC       PSTRNG
04EC 283C0000.         MOVE.L   #0,D4
04F2 2A3C0000.         MOVE.L   #0,D5
04F8 1800              MOVE.B   D0,D4
04FA A038              DC       OUT5D
04FC 4E75              RTS

04FE 000002B6          END      START
```

Beginner's Corner

by Ron Anderson

Last time we talked in general about the computer. Assuming those who are reading this all have a computer whose processor is the 68008, 68000, or 68020, you will all be interested in learning more about that microprocessor family. Unfortunately, we have to start with the dull and dry subject of number representation. In virtually all of the microprocessor based computers, numbers are represented inside the computer in binary notation. That is, each memory bit must be on (representing 1) or off (representing 0). (Remember the movie Tron with the memory bits floating around saying "yes" or "no")? Just as in decimal notation as we proceed to the left in a number, the digits represent successive powers of ten, in binary notation they represent successive powers of 2.

Decimal

1 = 10^0 (ten to the zero power = 1)
10 = 1 times 10^1 + 0 times 10^0 = 10 + 0
23 = 2 times 10^1 + 3 times 10^0 = 20 + 3

Binary

1 = 1 times 2^0 = 1
10 = 1 times 2^1 + 0 times 2^0 = 2
100 = 1 times 2^2 + 0 times 2^1 + 0 times 2^0 = 4
101 = 1 times 2^2 + 0 times 2^1 + 1 times 2^0 = 5

Note first of all that any (non-zero) number to the zero power (N^0) equals 1. Any number to the first power equals itself. 2^1=2, 10^1=10. 10^2 = ten squared = 100. 10^3 = ten cubed = 10 times 10 times 10 = 1000. In binary notation the 0 denotes the absence of the power of 2 whose place it occupies. A 1 denotes the presence of that power of 2. The value of each "1" in a binary number is twice that of its neighbor to the right:

1 = 1 decimal
10 = 2 decimal
100 = 4 decimal
1000 = 8 decimal
10000 = 16 decimal

If the number has multiple 1s in it, the values ADD.

11 = 3
101 = 5
111 = 7
1111 = 15 (8+4+2+1) etc.

Now when we get to representing large numbers they get very tedious to look at:

1000000000000000 = 32768 decimal

It is hard to sort out all those zeros. Computer folks decided that they could combine such a number into groups of four binary digits and use a shorthand notation:

BINARY	DECIMAL	HEXADECIMAL
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Some of the early computers used Octal notation, but just for now, we'll skip that. Notice that hexadecimal runs out of our standard number symbols at 9 and then uses the first six letters of the alphabet for the remaining numbers. What is the use of this notation? 1010 0111 becomes A7. 1000 0000 0000 0000 becomes 8000. To indicate that numbers are hexadecimal, Motorola has always used a dollar sign preceding the number as in $A7. Since a hexadecimal number could possibly contain only the symbols 0 through 9, the number base must be specified, and the $ is always the key.

Let's talk a minute or two about how computers handle data, and data sizes. One Binary digIT (represented by 0 or 1) is called a BIT. A number of length 8 bits as in 1000 0000 is commonly referred to as a BYTE. Most computers allow access to memory in one-byte "chunks". A byte is just right for representing a single character of text. 8 bits can represent unsigned numbers from 0 to 255. That sounds like too much to represent characters. The ASCII codes for characters occupy the first half of those values 0 to 127. The values 0 to 31 represent "control codes" including Carriage Return (decimal 13, Hex $0D), Linefeed (Decimal 10, $0A), and Formfeed (decimal 12, $0C). Note that a byte may nicely be represented by two Hexadecimal digits. The 8th bit can be used as a "parity" bit, a piece of "redundant" information for error detection, (more on that later) or as a flag to a printer to make that character italic, or as a means of representing graphics characters (chunks of single or double ruled boxes, smiley faces, club, spade, diamond, heart symbols, etc.

The 68000 also allows handling of larger numbers directly. You can access and manipulate 16 bit "WORD" length chunks of data (represented by four Hexadecimal digits, 16 bits) and "long" data (represented by eight Hexadecimal digits or 32 bits). Word sized data generally is used to represent integer numbers. Numbers in the range of -32768 to 32767 can be represented. Sometimes Words are used to represent "unsigned" numbers (always considered positive) and can then represent numbers from 0 to 65535. Longs can represent signed numbers in the range of about +/- 2,140,000,000. More about this below.

10100001: What is the decimal value? If you count places, doubling each time, you will get 1+32+128 = 161. What is the hex. value? $A1. Is there an easy way to get from the hex value to the decimal value? Yes, each digit is worth the next succeeding power of 16. That is, the lowest digit's value is multiplied by 1 (16^0), the second digit is multiplied by 16, the third by 256, etc. $A1 therefore is 10 times 16, plus 1, or 161.

The 68008 has 20 address lines (or pins on the IC that carry memory addresses). The largest address that can be accessed is $FFFFF, which is 1048575 decimal. Remembering that 0 is a valid address, the 68008 can access 1048576 addresses, just over one million bytes, commonly called 1 megabyte. The 68000 has 23 address lines (plus one internal line, for a total of 24), so its maximum address is $FFFFFF or 16 megabytes of memory. Obviously we would not like to represent addresses on a 68000 as a string of twenty-four 1's and 0's. The hexadecimal notation is useful. The 68000 Assembler can accept numbers in decimal, binary, or hexadecimal. There are reasons for using each of these notations at different times.

Well, we got through that. Now let's look at the 68XXX processor itself. This discussion won't be complete by any means, but it will get us started writing a program and running it. The 68XXX has 8 Data registers and 8 Address registers; the diagram at the right shows the inside of the 68000 as seen by a programmer. A "Register" is simply a place that can hold a number value. If you will, it is a special memory location built right into the processor. All of these registers are 32 bits "wide". That is, they can hold a Long number, 8 hexadecimal digits from 0 to $FFFFFFFF. The registers can therefore hold numbers from 0 to around 4,290,000,000 decimal, or "signed" numbers from around -2,140,000,000 to +2,140,000,000. Perhaps it is time to mention that so far, and for the time being, we are dealing only with integer numbers (whole numbers), numbers without decimal fractions such as 3.14159265398. Such numbers are called "Real" numbers or "Floating Point" numbers. We will get into those much later. The eight data registers are named D0-D7, and the eight address registers are named A0-A7. Since each register can hold 32 bits, the bits are numbered from bit 31 at the left to bit 0 at the right.

Suppose we start out and write a simple program to add 5 and 7. The most used instruction for the 68000 assembler is MOVE. Instructions operate on "Bytes" (8 bit quantities), "Words" (16 bit quantities) or "Long Words", 32 bit quantities. The letters B, W, and L are used as suffixes for instructions to tell the assembler how big a "chunk" of data to move, Byte, Word or Long. Some examples:

MOVE.W D7,D5

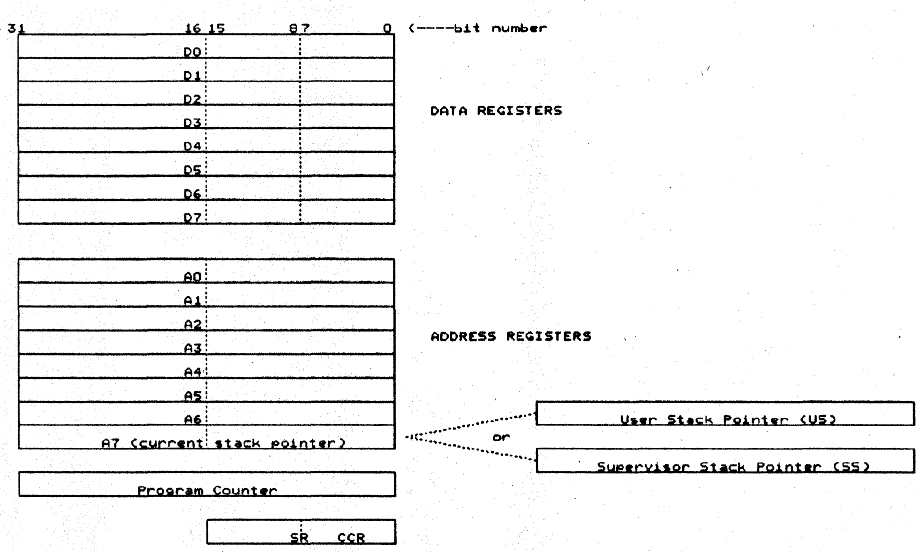
This instruction moves a word (16 bits), the rightmost or low order 16 bits of the 32 bit contents of D7, to D5. The upper 16 bits of D5 will remain unchanged, as will all of D7.

MOVE.L D7,D5

This instruction moves the entire 32 bit contents of D7 to D5. When the contents of a register are moved to another, the first or source register remains unaltered. Thus if D7 contains $789ABCDE and D5 contains $12345678, after the MOVE.W instruction above, D7 would still contain $789ABCDE, but D5 would contain $1234BCDE. After the MOVE.L instruction, both registers D7 and D5 would contain $789ABCDE. We can also move a BYTE. Byte values can represent signed numbers from -128 to +127. Thus we can add 5 to 7 using Byte values. It is possible to move a predetermined number into a register by preceding it with the # symbol. For example, the following instructions put the number 5 into register D7, the number 7 into D5, and then add the number from D7 into D5:

```
MOVE.B #5,D7
MOVE.B #7,D5
ADD.B  D7,D5
```

The ADD instruction must also specify the size of the data on which it is to operate (the OPERAND). The last instruction adds the byte contents of D7 to D5 and puts the result in D5. The second register specified is always the recipient of the result, be it a move, an add, a subtract, a multiply, a logical AND or OR, or whatever. The # (some people call it a number sign, some a pound sign and some a *sharp*), is read "immediate" in assembler language programming. "Move



What the programmer sees inside the 68000 dot B immediate 5 comma D7" is how I would read the first line of the three above. Immediate refers to the fact that the data value follows the instruction immediately. That is, the value 5 is coded right into the machine language instruction by the assembler.

All this is nice, but how do we peek to see that D5 actually contains the value decimal 12 or Hex $0C? Well, we can write a program to translate the $0C into decimal digits 12, change them to the ASCII codes for "1" and "2" and output them to the terminal, or we can take advantage of SK*DOS, our operating system. The Assembler has an instruction called DC, for "declare Constant", but also think of DC as *DOS Call*, because that is what we also use it for. Among the nice things that are in the operating system manual are the descriptions of the "system calls" that can be used by a programmer. A little look through section 10 of the manual will find descriptions of the DOS calls OUT5D and OUT2H. One outputs a number in decimal and the other in HEX. We have a slight problem in that OUT5D outputs a 16 bit or WORD value, so our Byte arithmetic needs to be extended. Let's try the following:

```
START MOVE.W #5,D7
 MOVE.W #7,D4
 ADD.W D7,D4
 CLR.L D5
 DC.W $A034  PCRLF
 DC.W $A038  OUT5D
 DC.W $A034  PCRLF
 DC.W $A03A  OUT2H
 DC.W $A034  PCRLF
 DC.W $A01E  WARMST
 END START
```

Now if you have an editor, type the above in just as you see it and save it as a file named ADDTEST.TXT. It is very important that the word START begin in the very leftmost column when you type the program in, and that all the other lines begin in the second column or farther. It is also important to put spaces exactly where they appear in the above listing and nowhere else. The reasons will become clear later. Now assemble it:

```
SK*DOS:   ASM ADDTEST -B
```

You will see the following listing appear on your terminal:

```
001 00000000 3E3C0005  START MOVE.W   #5,D7
002 00000004 383C0007        MOVE.W   #7,D4
003 00000008 D847            ADD.W    D7,D4
004 0000000A 4285            CLR.L    D5
005 0000000C A034            DC.W     $A034  PCRLF
006 0000000E A038            DC.W     $A038  OUT5D
007 00000010 A034            DC.W     $A034  PCRLF
008 00000012 A03A            DC.W     $A03A  OUT2H
009 00000014 A034            DC.W     $A034  PCRLF
010 00000016 A01E            DC.W     $A01E  WARMST
011 00000018 00000000        END      START

No Syntax Error(s)
```

The leftmost column is simply the line number in the program. The assembler numbers the lines sequentially. The second column is the MEMORY ADDRESS at which each instruction will load when the program is run. (It is not quite that simple, but that is OK for now). The third column is the actual code that the assembler generated in hexadecimal notation. Notice that the first two instructions each generated 4 bytes of code (eight hex digits) and the rest just two bytes each. I've used the hexadecimal values for the system calls right out of the manual, and included the names of the system calls after those values as "comments". To clarify, the word START is in the next column and it is a LABEL. After that comes the operator column that contains the instruction MOVE or DC or ADD. Then come the operands or operand, and anything after that is a comment. Note that at this point you have assembled the program with no output but a listing of the program to your terminal. In order to run it you must use a different assembler option to generate an executable output file, which we will do shortly.

You might notice that the assembler output listing is "expanded". That is, though the labels in the input file start at the first column and the operators (such as MOVE) at the second, the assembler has given the labels seven columns of space in the output listing *(Editor's note: I've eliminated some of the extra spaces and zeroes so the code would fit on the page.)* This assembler and most others allow you only 6 characters for a Label. Some allow you to use more, but only the first 6 are significant to the assembler. A few allow you to use very long labels if you want to do so. If you like, you can tabularize the input listing as well. I like to keep the input listing short so the files are small, since they are then read faster by the assembler. I generally make an output listing if I want to study the program to look for bugs or possible improvements.

I need to mention that the .W suffix on instructions is not needed. If the suffix is left off, .W is assumed. .L or .B are never assumed by the assembler. I generally use the .W so that all instructions have a suffix. It is much easier to find one that is wrong if you do this, particularly when you are beginning. Otherwise a missing .B is not obvious and you will hunt long and hard before it dawns on you what the problem is.

Let's make our program more readable. Rather than use the hex codes for the SK*DOS calls, we will define them in terms of the names of the calls:

```
* Program to add two numbers
 NAM ADDTEST2
* System Equates
PCRLF EQU $A034     These tell the assembler that
OUT5D EQU $A038      ...the name and the hex number
OUT2H EQU $A03A      ...stand for the same thing
WARMST EQU $A01E
*
START MOVE.W #5,D7
 MOVE.W #7,D4
 ADD.W D7,D4
 CLR.L D5
 DC PCRLF            Here we use the names, not
 DC OUT5D             ...the hex values
 DC PCRLF            Note the use of DC to mean
 DC OUT2H              ..."DOS Call"
 DC PCRLF
```

```
DC WARMST
END START
```

I haven't explained a couple of things here, and there are a few new ones. First of all, any line that starts with * in the first column is assumed to be a comment line by the assembler. It is passed on through to the output listing, but it doesn't generate any code whatever. The word START is a label and you will note a couple of new things. NAM is an "assembler directive" that says that what follows is the name of the program. If you print out a listing of the program using the assembler, the name will appear on the top of each page. The last line of the program is END START. END is the assembler directive to say that the program has ended. START, following it, is a special feature of Motorola assemblers. It tells the assembler to indicate that when the program runs, it is to start at the label START. It does this by writing a special instruction sequence to the program disk file at the end of the code. The label is unimportant. It could just as well be GOBBLE or XYZ (as long as it matches the actual label of the first instruction). You should see, however, that it makes the program easier to read if you use a meaningful label.

Oh, yes, I forgot to mention the last actual instruction in the program. When we run it (we'll do so momentarily) the instruction DC WARMST tells the processor to jump back into the operating system command processor loop and wait for another instruction. OK, now let's assemble the new version again with the -B option and check to see that there are no errors. If not, assemble it once more:

```
ASM ADDTEST2 -L
```

You won't see anything on the screen until the assembler is finished, when it will tell you that there were no errors. That is because the -L at the end of the command tells the assembler to assemble without a Listing (the -B earlier told it to assemble without writing a Binary or machine language file on the disk.)

The program will assemble and write an output fiie to your present directory. Assuming you are working on a floppy disk, or in the root (main) directory of a hard disk, CAT ADD will show you ADDTEST2.TXT, your "source" file, and ADDTEST2.COM, your "object file". This file is "executable". That is, you can run it. At the SK*DOS prompt, type ADDTEST2 and then press the Carriage Return or Enter key). You should see:

```
   12
0C
```

The 12 is indented because 5 characters were printed. Depending on what is in D5 at the time the OUT5D routine is used, SK*DOS may precede the 12 with either three spaces or three zeroes; we set D5 to zero, so the leading zeros were changed to spaces. The 0C is at the left of the screen because the PCRLF call caused a Carriage Return and Line Feed to be sent to the terminal, and there were no leading characters with OUT2H.

ADDTEST2.COM is now a command on your system disk. Simply type ADDTEST2 and the program will run.

Well, that is just a little start at assembler programming and using the SK*DOS system calls. There's a lot more to learn. Change the ADD instruction to SUB and you will get 7 minus 5 as a result. Remember that you don't just change the source program - you have to assemble it. Note that you put 5 in D7 and 7 in D4. The SUB instruction subtracts the value in the source register from the value in the destination register and puts the result in the destination register. To subtract 3 from a register you would say:

```
SUB.W #3,D4
```

Of course, in the above program we could move 7 into D4 and use SUB.W #5,D4 and save an instruction. One thing that becomes very clear when you start programming in assembly language is that there are many ways to accomplish the same thing. We've only used three Data registers in our program, and have not used any Address registers at all, at least explicitly.

Perhaps this is more than enough to digest for this time. Next time we will continue to expand this demonstration program and learn new assembler instructions and techniques.

(Editor's note: for a review of how to use the editor and assembler, see the last issue, number 8, of 68 News.*)*

Speed up SK*DOS Booting

by Peter A. Stark

When SK*DOS boots on a PT68K-2 system, it checks how many (if any) hard drives are installed. Because of some peculiarities of the hard drive controller, this takes about 20 seconds, during which we all stare at the computer, wishing it could be speeded up. Well it can!

Starting with version 3.0, SK*DOS has a location which tells it how many hard drives to look for. This location is in address $1152; SK*DOS is normally supplied with an 02 in that location, so SK*DOS looks for two drives. By changing this byte to an 01 if you have one drive, or to 00 if you have only floppy drives, you can speed up booting a lot. Here is how.

First, using your editor, prepare the following text file; call it FASTBOOT.

```
 ORG  $1152
 DC.B 1        (or 0 if you have no hard drive)
 END
```

(Make sure to leave at least one space at the beginning of each line.) This short program simply tells the system to define a byte constant (DC.B) equal to a 1 (or 0), and put it into memory at location $1152 hex.

Now assemble it with the command ASM FASTBOOT into a file called FASTBOOT.COM. Although this has a .COM extension and therefore sounds like a command, it is not actually a full program and cannot be run. Next, rename your SK*DOS system file with the command

```
RENAME SK*DOS.SYS SK*DOS.TWO
```

and then make a new SK*DOS system file by appending the FASTBOOT file to the end of this original SK*DOS file, using the command

```
APPEND SK*DOS.TWO FASTBOOT.COM SK*DOS.SYS
```

The result is your original SK*DOS file which has location $1152 set to a 02, but appended to it is the FASTBOOT file, which consists of just one byte - a byte which overlays the 02 in location $1152 and substitutes an 01 (or 00, depending on your situation). The next time you boot from this version, you will find that SK*DOS boots within just a second or two. Just save the original file, so if you do get more hard drives, you can easily go back.

From: Star-K Software Systems Corp.
P. O. Box 209
Mt. Kisco, N. Y. 10549

Address Correction Requested

BULK RATE
U. S. Postage
PAID
Mt. Kisco NY 10549
Permit No. 197

To: