PROMAL

Public Domain Library
Commodore Disk Number 1

PDC - 001

Systems Management Associates
3325 Executive Drive, P.O. Box 20025
Raleigh, North Carolina 27619

Contents of Commodore PROMAL Public Domain Disk #1

PLEASE NOTE: All public domain disk materials are contributed works. SMA is only serving as a clearing house for these materials as a service to our PROMAL customers. You may copy, use, and further disseminate all Public Domain Disk materials as you see fit. All materials are supplied "as is". SMA does not support these programs in any way. PLEASE DO NOT CALL with questions.

Macro assembler by C. Martens

ASSEM.(S,C) = the assembler. ASSEM.S also uses include files
SYM.S, ASSEMTABLES.S, UPSTR.S, EXP.S, GETMODE.S, EQUATE.S,
PSEUDOOP.S, INSTRUCT.S, MACRO.S, CODER.S.
Note: DYNO must be off when running ASSEM.C.
ASSEMDOC.T = documentation
DDUMP.(S,C) = program to dump a file in hex.
BLOAD.(S,C) = program to load object file made by ASSEM.C
EXAMPLE.(A,Z) = example program for ASSEM.C. EXAMPLE.Z is
the object program produced when EXAMPLE.A is assembled. EXAMPLE.A
also uses include file SYSTEM.M, a collection of macros.
MACROT.(S,C) - learning aid for macros. Explained in ASSEMDOC.T.
EXPT.(S,C) - learning aid for expressions. Explaind in ASSEMDOC.T.

Assembler capabilities include macros, include files, conditional assembly, arithmetic and boolean expressions. Output file is a PROMAL sequential file beginning with 2-byte load address, with remaining bytes being the assembled code. This object file can be loaded by the BLOAD program above, but cannot be loaded by the PROMAL EXECUTIVE commands, because it does not conform to the definition of a PROMAL relocatable object module (as described in the RELOCATE utility comments). It also cannot be loaded with the MLGET function, because it is not type OBJ (for Apple) or PRG (for Commodore).

However, the RELOCATE utility (on the PROMAL 2.0 END USER DISK) can easily be modified to convert the Martens' assembler output to a relocatable PROMAL module. The following minor changes to the RELOCATE source code are required. Please save the original version of the source! The lines beginning with question marks are for conditional compilation. The following modification works for the Commodore version:

In file RELOCATE.S, in PROC OPEN_FILE, replace the lines:

```
IF MODE <>'W'
?A
  HANDLE=OPEN( NAME,MODE,TRUE)   ; Non-Promal file
?
?C
  HANDLE=OPEN( NAME,MODE,'P')    ; PRG file
?
ELSE
  HANDLE=OPEN( NAME,MODE)    ; PROMAL output file
```

with the single line:

```
HANDLE=OPEN( NAME,MODE)   ; Open file
```

Compile your new version of RELOCATE. You are now ready to assemble your program twice with different .ORG statements as described in Appendix I of the PROMAL manual and use your new RELOCATE on the resulting output files.

PS: If you use Martens' BLOAD utility instead of making the object code into a PROMAL module, and you execute the program from the EXECUTIVE with a GO command, please note that your assembly program MUST save the word at location $0065 (Apple) or $003A (Commodore 64) on entry and restore it before exiting with an RTS. If you don't and you call any PROMAL library routines inside your assembly program, your program will not return to the EXECUTIVE. The same thing applies if you enter the machine language program by using PROC JSR.

Disassembler by Steve Vermeulen

DIS.(S,C) = the program

Disassembles instructions in a specified range of memory addresses. For example, the executive command "DIS 1000 1400" disassembles memory data within the given range of addresses. See source file for more usage information.

Disk Fixer program and documentation of disk structures by A. Ryan

DISKFIX.(S,C)	The fixer program. It uses the following includes: LIBRARY.S, C64_EQUATES.S, JSRDEF.S, GETCHR.S, CSRUPDATE.S, BUILDSCREEN.S, SCANKEY.S, ADDR.S, EDITCHR.S, EDITEXT.S, INCHR.S, BLOCK.S, RWBUF.S, FETCH.S, INIT.S, LINK.S, SCTRUPDATE.S, WRITE.S, PUTBUF.S, TERMINATE.S, WINDOW.S, BPRNT.S, FDJOB.S, FDERR.S, SCTRCHK.S, DSTAT.S, INBUF.S, PRINT.S, OPN.S
CHANGE.(S,C)	Locks/unlocks a file. Execute CHANGE filnam L to lock, and CHANGE filnam U to unlock. See documentation and commentary in CHANGE.S.
STAT.(S,C)	A program to issue a command to the 1541 disk drive and see the completion status. See documentation and commentary in CHANGE.S.
TIME.(S,C)	Demo for C64's real time clock. Initialize it with TIME HH MM SS. Subsequent TIME executions with no arguments show the current time as "hh:mm:ss".
Documentation	The included hardcopy documentation explains the 1541 disk structures, how commands are issued to the drive, and how to use DISKFIX. DISKFIX allows you to display any sector in an easy-to-recognize format and to easily make changes.

DISKFIX-Part 1

By A. Ryan

This series of three articles describes the DISKFIX system that permits you to manipulate data directly at the track and sector level of the COMMODORE-1541 Disk drive. The primary use of this utility is to repair or rescue data that has become otherwise unreadable by the 1541 DOS, typically as a result of an accidental SCRATCH or PROMAL DELETE command. In addition, discs whose files have become "poisoned" as a result of a variety of circumstances can often be rescued with careful editing of various data on the disc.

Since this utility bypasses most of the native 1541 DOS, it is extremely powerful, and in unskilled hands is capable of seriously damaging the data in your files. In skilled hands however, it is possible to retrieve data that would be considered as totally unrecoverable by any other means. Thus, in order to ensure that you are in a position to reap the maximum benefit from such a tool, it is necessary that you understand three pieces of information;-

[1] How data is stored and organised on the disc;

[2] How the 1541 is controlled;

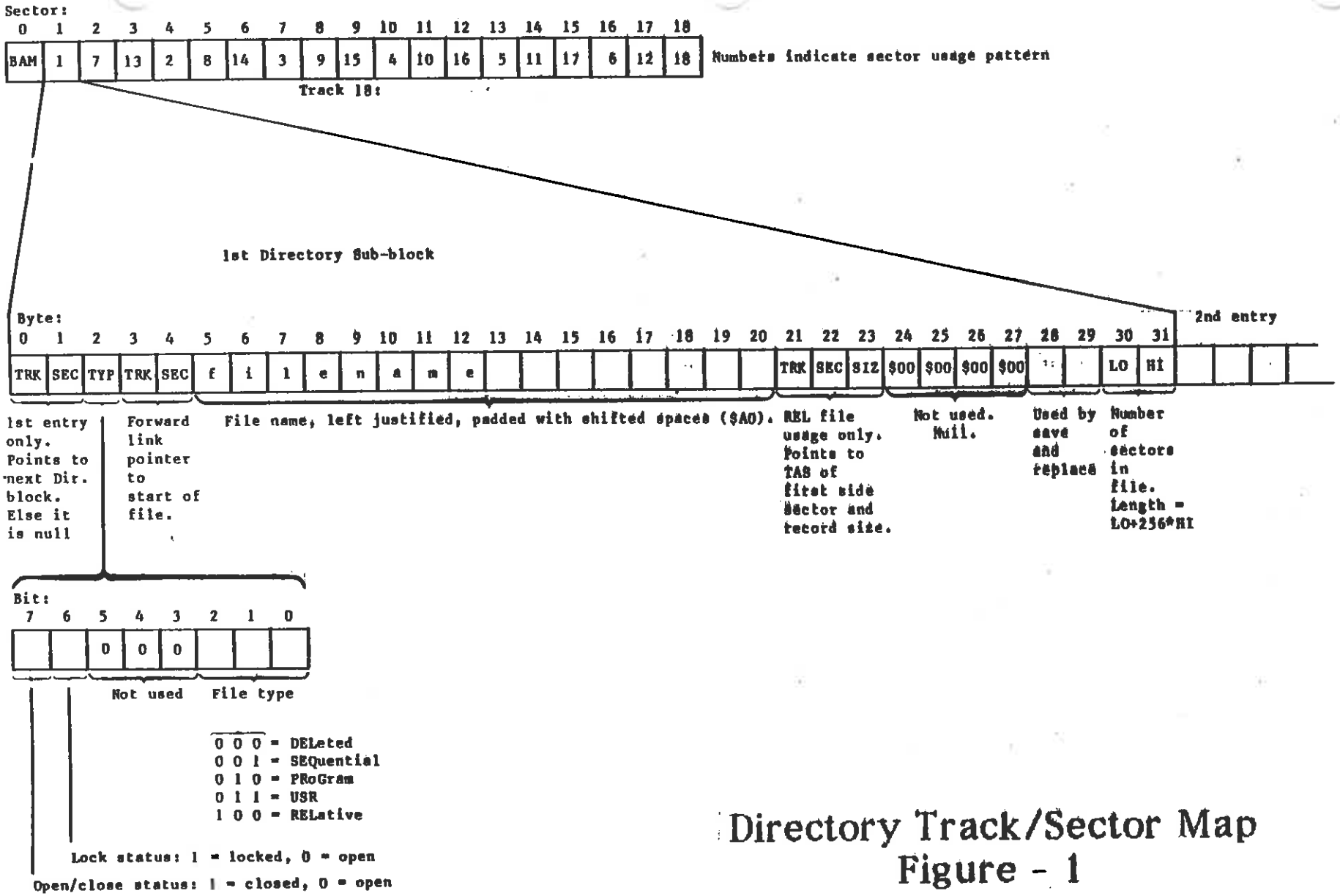
[3] How DISKFIX works, and how to use it.

For this reason this article is divided into 3 sections covering each of the above areas.

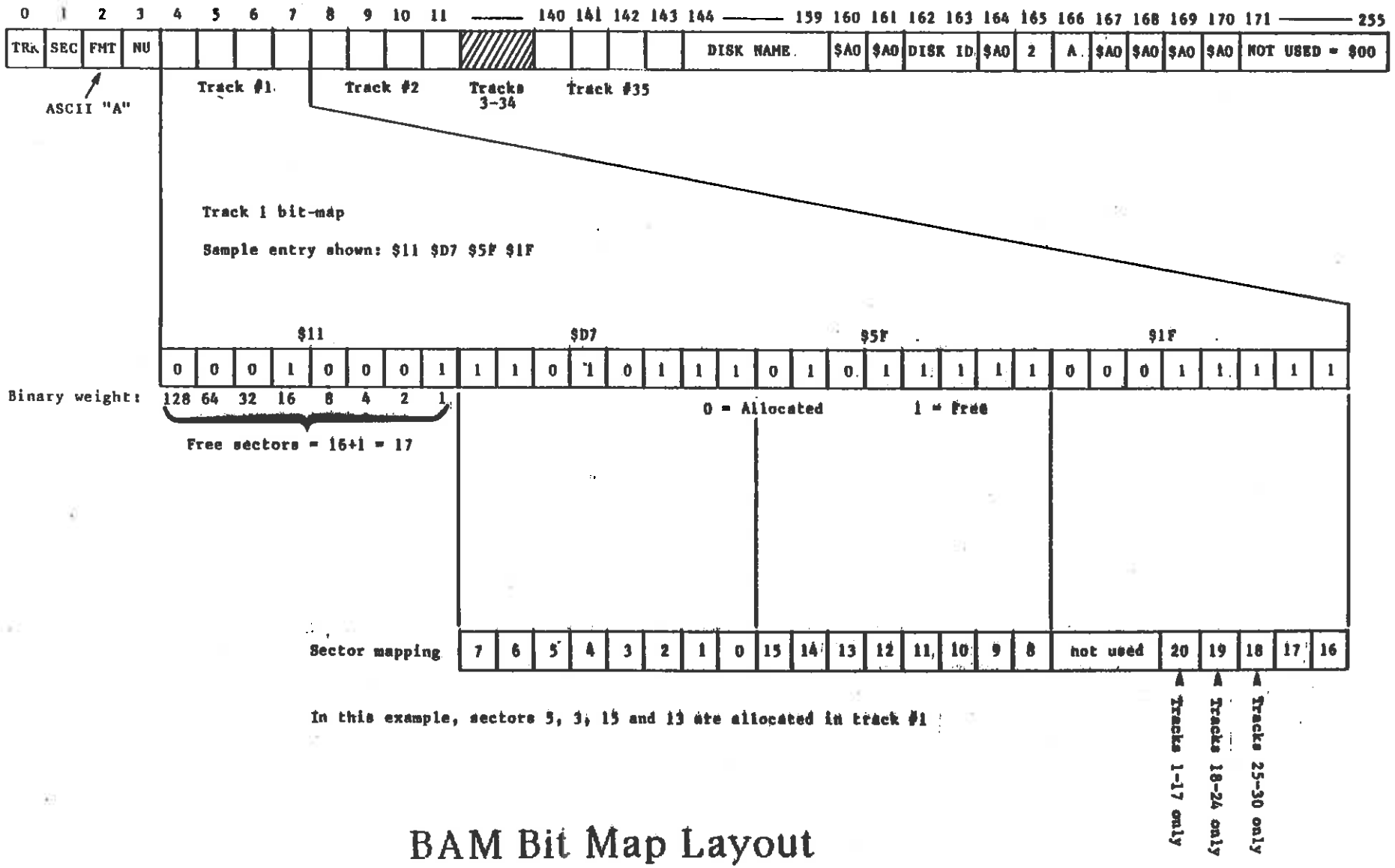
In this portion, we shall examine how data is stored and organised on the disc itself. Once the disc has been formatted, (for example, by means of the command: OPEN 15,8,15,"N0:TEST-DISC,ZZ) it consists of 35 concentric rings or TRACKS numbered from TRACK 1 at the outermost edge of the disc, to TRACK 35 at the centre. Each track is further sub-divided into segments called SECTORS, and the number of SECTORS per TRACK vary from 17 at the innermost tracks to 21 at the outermost. Each SECTOR can contain a maximum of 256 bytes of information. Of these 256, the first 2 are reserved for use by the system, leaving 254 bytes free to store your data. Table 1 shows the distribution of SECTORS and TRACKS for the 1541.

TRACK NUMBER	SECTORS
1 -- 17	21
18 -- 24	19
25 -- 30	18
31 -- 35	17

Table - 1



Directory Track/Sector Map
Figure - 1



BAM Bit Map Layout
Figure - 2

The reason for the varying number of sectors per track is to minimise the variation in recording density between the inner and outer tracks, in an attempt to increase the reliability of the drive. Most other disc systems, ie, APPLE, TANDY, etc, adopt the simpler approach of a constant number of sectors per track.

To give you some idea of the dimensions with which we are concerned, on the 1541, at the outermost tracks the bit density is about 4000 bits/inch whilst on the innermost tracks it is about 6000 bits/inch. To put it another way, in the distance represented by the diameter of a human hair, at the innermost tracks this distance will contain about 8 bits or 1 byte. You can thus understand that a small piece of dust, dirt or smear of a fingerprint can obliterate quite a lot of data!

In order for the Disk Operating System (DOS) to be able to locate a file of data, there has to be an index of file names, usually called the DIRECTORY. In addition, there must also be a 'map' showing which sectors of the disc have been allocated. In the 1541, this 'map' is called the BLOCK AVAILABILITY MAP, and both the BAM and the DIRECTORY are stored on TRACK 18.

This track was chosen to minimise the number of head movements necessary to access any particular file. Track 18, from Table - 1, has a maximum of 19 SECTORS available. Of these, Sector 0 is reserved for the BAM, whilst Sectors 1 - 18 are available for the DIRECTORY. The organization of Track - 18 is illustrated in Figure - 1.

Ignoring for the moment the BAM, each DIRECTORY sector is divided up into eight 32 byte sections that store a single entry. Hence, since there are 18 sectors available for the DIRECTORY, and each sector can hold 8 file records, the 1541 can have a maximum of 144 files on each disc.

For the very first entry in each sector, bytes 0 and 1 are the FORWARD LINK POINTER to the next sector, (this will be further explained later) whilst bytes 2 through 31 contain the actual directory data. In the second and subsequent 32 byte segments, bytes 0 - 1 contain $00. Byte 2 of each directory entry is the FILE TYPE DESCRIPTOR, and its mapping is also shown on Figure - 1.

Bits 0,1, and 2 of the FILE TYPE DESCRIPTOR are used to identify the 5 COMMODORE file types. Notice that since 3 bits are allocated for this function, in principle it is possible to map 8 unique file types. By using the un-mapped 3 codes it is possible to produce a number of bizarre file types, but in practice these 'un-mapped' types are of little practical value.

Of the remaining 5 bits in the byte, only bits 6 and 7 are utilized by DOS. Bit 7 denotes the current status of the file; a '1' signifies that the file has been properly 'closed', whilst a '0' signifies that the file is currently still open.

Improperly closed files are one of the root causes of 'poisoned' discs, and as will be explained later, must be treated with great care if you are to avoid further poisoning of the disc.

Bit 6 is an undocumented feature of the 1541 DOS, and signifies the LOCK/UNLOCK status of the file. By setting this bit to a '1' you can prevent DOS from erasing this file. (A future article will present and explain the PROMAL source code for 3 routines, LOCK, UNLOCK, and CHANGE that take advantage of this feature to enable you to prevent your files from being DELETED by PROMAL)

Bytes 5 through 20 are the left justified file name padded out with the $A0 Shifted-Space character.

Bytes 21 through 23 are used in the RELative file definition, and they will be explained later, for other than RELATIVE file types, they are $00. Similarly, bits 24 through 27 are also unused by DOS.

Bytes 28 and 29 are ordinarily set to $00, but are used dynamically by the SAVE & REPLACE feature of CBM-DOS Version 2.6 to temporarily store the FORWARD LINK POINTER during the SAVE portion of the new file data. This data gets transferred to its normal location, bytes 3 and 4 of the 32 byte record, at the conclusion of the operation.

Finally, bytes 30 and 31 are the file length descriptor word in normal 6502 Lo-byte/Hi-byte format.

The FORWARD LINK POINTER is the means by which DOS links the various sections of a file stored on the disc. Bytes 3 and 4 of each 32 byte file record contain this pointer, byte 3 points to the TRACK where the next segment of the file is located, whilst byte 4 points to the SECTOR location.

This mechanism is used for all the files, and bytes 0 and 1 of every sector of actual file data contain these pointers. When the last sector is written to the disc, byte 0 is set to $00 to indicate to DOS that this is the last sector of the file. Remember, TRACKS are numbered from 1 - 35, thus there is no TRACK 0. The value contained in byte 1, which would ordinarily mean the next sector location, now indicates the number of valid bytes in this sector, and will have a value from 3 to 256.

At this point we can now explain the construction of the BLOCK AVAILABILITY MAP or BAM. This is best done

pictorially, as shown in Figure - 2. Notice that, as usual, bytes 0 and 1 contain the FORWARD LINK POINTER, in this case for the DIRECTORY, and will normally point to TRACK 18 SECTOR 1.

Byte 2 contains the DOS format identifier, byte 3 is unused, whilst bytes 4 through 143 are grouped in 4 byte records that map the allocation table.

Bytes 144 through 159 contain the left justified disc name padded out with shifted spaces, bytes 160 and 161 also contain the $A0 shifted space character.

Bytes 162 and 163 contain the disc ID characters, byte 164 contain $A0, bytes 165 and 166 contain "2A", bytes 167 through 170 contain $A0, and bytes 171 through are unused, and contain $00.

The mapping of the allocation table is quite straightforward. The first byte of the group of 4, i.e byte 4 of the sector, byte 8, byte 12 etc, contains a value representing the number of free sectors in the track represented by the 4 byte group. The remaining 3 bytes represent a straight bit map of the allocated sectors. A '1' means the sector is free, whilst a '0' means that it is allocated. It is easier to illustrate than describe, and I suggest that you study Figure - 2 to gain a clear picture of the techniques used. Notice that the first 4 byte group maps TRACK 1, the next 4 byte group TRACK 2, and so on.

The only unusual sequence is the sector mapping within the group. Again, it is much easier to understand from a picture rather than a textual description, so please refer to Figure - 2.

Whilst the mapping of the BAM is interesting from an academic standpoint, in practice there is very little need to 'tinker' with the data of the bit map. If you need to re-construct a bit map from a damaged disc, then use DISKFIX to recover the data, and when leaving DISKFIX select the 'update BAM' option, which will perform this chore for you.

At this point a word of warning is appropriate. If you use DISKFIX to rescue a file that was SCRATCHED or DELETED, then DOS will have de-allocated the sectors from the BAM. Unless you force a re-construction, or perform it yourself, although the file can now be recovered from the disc, the next WRITE operation will, in all probability, overwrite one or more of your files sectors, and you will commence to 'poison' the disc again! This point will be covered in more detail when we get to part 3.

We have already seen that there are only 5 types of files recognized by COMMODORE DOS, namely, DELeted, PRoGram, SEQuential, RELative, and USeR. Of these 5, the PRG and SEQ are actually the same types of file structures, whilst even

the REL type has considerable similarities. The DEL file is simply a deleted or scratched file, rather than a separate file type, and retains all the characteristics of its previous existence. We shall commence therefore with the most fundamental file type, the sequential file.

In a sequential file, the data you wish to store is simply laid down, 254 bytes per sector, one sector at a time. In order to manipulate this type of file, the entire file has to be read into the machine, and any modifications made directly to memory before re-writing the entire file back to the disc. Since memory manipulations are extremely rapid in comparison to disc access times, such a file can be manipulated quickly. The limitation, however, is that the file cannot be larger than the available memory, and usually only a small fraction of the total available memory. This type of file is usually used for the storage of programs, text files, and other data that does not need to be retrieved from the disc on an individual record basis.

In the case of the 1541, the FILE TYPE DESCRIPTOR byte will be set to $81 (SEQ) or $82 (PRG) and the FORWARD LINK POINTER in bytes 3 and 4 of the 32 byte directory record will point to the TRACK and SECTOR that contain the first sector of the file. At this first sector, and at every subsequent sector save for the last, bytes 0 and 1 will be the FORWARD LINK POINTER to the next segment of the file. The last segment will be identified because byte 0 will be $00, and the contents of byte 1 will represent the number of valid data bytes in this sector.

If the sequential file in question is actually a program, (either BASIC or M/L) then the first 2 bytes of the FILE will contain the load address. Since the first 2 bytes of the SECTOR will contain the forward link pointer, it will be bytes 2 and 3 of the first sector of the file that will contain the load address. For example, all normal BASIC programs will start at $0801, thus byte 2 will contain $01 and byte 3 will contain $08, in normal 6502 Lo-byte/Hi-byte format.

If the sequential file simply contains data, then byte 2 and 3 of the first sector will contain data.

The next type of file file is the RELATIVE file, and its structure is, in reality, only slightly more complex. The object of using RELATIVE files is twofold:-

[1] To be able to reference more data than can be contained in memory;

[2] To be able to access any record within the file.

In order to achieve these objectives, the data to be stored must be organised into fixed length RECORDS, and that the maximum record length cannot exceed 254 bytes. Each

record is sub-divided into FIELDS, and a these are further sub-divided into CHARACTERS.

In order to be able to access any individual record of the file, DOS has to know where the record is located on the disc surface. This information is contained in the SIDE SECTOR file created by DOS when the file structure was originally set up. The SIDE SECTORS are simply a small sequential file containing a list of TRACK/SECTOR pointers.

If you examine Figure - 1 again, you will observe that bytes 21 through 23 are reserved for use by RELATIVE files. Byte 21 contains the TRACK, byte 22 the SECTOR, and these point to where the first SIDE SECTOR segment is located. As usual, bytes 0 and 1 of this sector will point to the next and so on. Byte 23 contains the record size that was defined when the file was initially created.

The mapping of the SIDE SECTOR file is:-

```
Bytes  0 -  1   Forward Link to next side sector
Byte   2        Number for this side sector
Byte   3        Record Length
Byte   4 -  5   T & S for side sector 0
Byte   6 -  7   T & S for side sector 1
     :
     :
     :
Byte  14 -  15  T & S for side sector 5
Byte  16 -  17  T & S for data block  1
Byte  17 -  18  T & S for data block  2
     :
     :
     :
Byte 254 - 255  T & S for data block 120
```

As you can see, since there is provision for a maximum of 6 side sectors, and each side sector can refer to up to 120 data blocks, the maximum number of sectors that can make up a file is 720. Note that this does not mean 720 RECORDS, but rather 720 SECTORS. Since a blank disc can only hold 664 SECTORS, in practice one cannot reach this limit. The maximum number of RECORDS is 65535, and since this implies a RECORD length of 2 bytes maximum in order to stay within the bounds of the 163K bytes of disc space, this is also unlikely to be reached. As a final note concerning RECORD size, you should make the size equal to 1 + (number of chars in a record). This allows for the CR character that terminates a record.

Records themselves can contain whatever data you desire, and in order to conserve file space, most files will concatenate the fields into a record, and separate the fields when the individual records are read. If you wish to avoid this, then you can use a field separator character, but this will increase the field size, and in addition, this

separator character may not be used within a field as part of the data. In general, a CR or an ASCII ',' is the normal field separator.

Since the 254 byte maximum limit on RECORD size is only factored by 1,2,127, or 254, in most cases records will span sector boundaries. This may well provoke a 'bug' in the way DOS updates records. This bug can be avoided by ALWAYS positioning the record pointer both BEFORE and AFTER writing to a record. If your record sizes are one of the factors, of which only 127 and 254 are very practical, this bug is not triggered.

This explanation is only a very brief overview of RELATIVE file fundamentals, sufficient for you to understand how they are stored on a disc. To cover the subject in depth would require an article in its own right, and is really beyond the scope of this tutorial.

This article has attempted to show you how data is organised on the disc surface. In Part II we shall examine how the 1541 is controlled from the hardware and software aspects.

DISKFIX-Part 2

In Part I we briefly skimmed over the way data is stored on the disc, and how the Disc Operating System places an index onto Track 18 to enable any file to be recovered. In this portion we shall examine how the mechanics of file storage and retrieval are accomplished.

The 1541 is somewhat unique in that it does not use a conventional VLSI chip for the disc controller functions. As far as I am aware, only the APPLE disc drive shares this distinction. Instead of a VLSI Floppy Disc Controller, the 1541 uses a 6502 microprocessor, equipped with 2K of RAM and 16K of ROM firmware to carry out the twin tasks of Serial Port Communications Processing and Floppy Disc Controller.

These twin tasks are performed on an interrupt driven time-shared basis, with interrupts occurring every 10 mSec. Normally, with the drive in an idle state, the Communications Processor (CP) is active scanning the serial port ATN line waiting for requests from the COMMODORE-64. Every 10 mSec, an interrupt generated from a timer in the Complex Interface Adaptor chip (CIA) will force the 6502 into the Floppy Disc Controller (FDC) mode. In this mode the 6502 will scan the TASK LIST looking for a valid task to perform. If no tasks are currently queued, then control is returned to the CP.

If a task is queued, then this task is performed, and at the conclusion a return code is put in the job queue representing the task status.

Let us consider a typical task, that of reading a sector of data. The DOS will put into the TASK LIST the job code for a SEEK of the desired track and sector. At the next interrupt, the FDC will attempt to execute this task by first comparing the current read head position with the target location, and stepping the head in or out the appropriate number of steps. When the head has settled, the search for sync characters commences. Assuming that this search is successful, the TRACK ID will be read. This ID, which is written during formatting, will contain the track number. If this number agrees with the desired track, then this fact is reported to DOS. If the number is not correct, then the difference between the actual and the desired position is used as the new step increment, and the head is moved again. In general, a SEEK will succeed on the first attempt. Then a READ job will be placed in the TASK LIST and the appropriate sector ID will be sought. If this search is successful, the data following the sector header will be read into a buffer area in RAM for processing by the CP.

This description is necessarily very brief, and does not take into account the many errors that can occur, but is sufficient for our immediate needs. The main point to grasp is that the 'servo-mechanism' of head positioning is essentially critically dependent upon the correct formatting of the disc. Unlike larger hard disc drives, the head is absolutely positioned on the disc surface. This means that it relies on the accuracy of the stepping motor to ensure that when the head is requested to be positioned over (say) track 5, that it is exactly over the track concerned. To show that it has arrived, the track is written during formatting with a track and sector ID code so that by merely reading this header the operating system can confirm that the head is positioned over the correct track. The stepping motor can step the head in 1/2 track increments, and it is one of the error re-try procedures to move the head 1/2 track from its present location and attempt to read the track again to attempt a recovery.

Before you leap for joy and think that by positioning the head in half track steps you can double the capacity of your drives, let me say that the width of the read/write head is rather wider that 1/2 track to ensure that sufficient overlap exists to take into account stepper motor positioning tolerances. Thus any attempt to write to every 1/2 track will ensure that when the head is positioned it will read a portion of the desired track and a portion of the adjacent track, thus thoroughly confusing the read electronics.

So far I have given you the essential 'flavour' of the tasks needed to satisfactorily read/write data. To go into the full detail is quite beyond an article of this nature, and regrettably, must be left for another day. I shall continue with a discussion of the nature of the RAM buffers and how they are used.

I have mentioned that the FDC scans the TASK LIST looking for a job to perform, this TASK LIST is an area of the 6502 RAM, and has associated with it a dedicated buffer area for each of the 5 usable positions in the TASK LIST. Table - 1 identifies the major locations and associated buffer areas.

TASK LIST	BUFFER ADDRESS	TRACK ADDRESS	SECTOR ADDRESS
$0000	0 $0300-$03FF	$0006	$0007
$0001	1 $0400-$04FF	$0008	$0009
$0002	2 $0500-$05FF	$000A	$000B
$0003	3 $0600-$06FF	$000C	$000D
$0004	4 $0700-$07FF	$000E	$000F
$0005	5 NO RAM	$0010	$0011

TABLE - 1

Thus, if the job code for a SEEK is placed in $0000, the

track number placed in $0006, the sector number in $0007, when the head has settled, and after a READ code has been placed in $0000, the data will be placed in buffer #0, from $0300 to $03FF.

Table - 2 lists the appropriate job codes.

JOB CODE	FUNCTION
$80	READ
$90	WRITE
$A0	VERIFY
$B0	SEEK
$C0	BUMP
$D0	JUMP
$E0	EXECUTE

TABLE - 2

After the tasks have been executed, the return code replaces the original job code, and Table - 3 identifies the meanings.

RETURN CODE	MEANING	ERROR CHANNEL CODE
$01	OK	OK
$02	READ ERROR, Header block missing	20
$03	READ ERROR, No sync character	21
$04	READ ERROR, No data block	22
$05	READ ERROR, Data checksum error	23
$07	WRITE ERROR	25
$08	WRITE PROTECTED	26
$09	READ ERROR, Header checksum error	27
$0B	READ ERROR, Disc ID mismatch	29

TABLE - 3

With the above knowledge, we are equipped to take control of the 1541 FDC directly.

In order to read and write to the 1541 directly, we need to open a DIRECT ACCESS channel to the drive. This is performed in BASIC by means of the statement:

OPEN file#, device#, channel#, "#"

To pass data to the 1541 via this direct access path, in BASIC one would use the form:

PRINT #15, direct access function code, channe.#, drive#, track, sector

The function codes that may be used are:

U1	BLOCK READ, read a data block into 1541 RAM
B-P	BUFFER POINTER, position pointer to any byte in 1541 RAM buffe
U2	BLOCK WRITE, write contents of 1541 RAM buffer to disc
M-R	MEMORY READ, transfer contents of 1541 memory to C-64
M-W	MEMORY WRITE, write to 1541 RAM
B-A	BLOCK ALLOCATE, set bit in BAM bit-map to put sector in-use
B-F	BLOCK FREE, reset bit in BAM bit-map to free sector
M-E	MEMORY EXECUTE, execute code in 1541 RAM/ROM
B-E	BLOCK EXECUTE, transfer code from disc and execute

Table - 4

At this point I propose to clarify the actual mechanism used within the COMMODORE operating system to exchange data with a peripheral device. It would seem that there is a certain amount of confusion in this matter, and neither the 1541 Users Manual nor any of the other ancillary documentation makes any attempt to clarify the subject.

As shown above, to open a direct access channel to the peripheral we use (in BASIC) a statement of the form:

OPEN file#,device#,channel#,filestring

Regrettably, the 'file#' does not refer to a file, and the 'channel#' does not strictly refer to a channel either! No wonder there is a great deal of confusion. It becomes even more confusing when we use machine-code to perform this chore, for, as we shall see in the final part, some of the syntactical conventions we used in BASIC simply do not work in M/C. Thus it is high time to put matters straight and explain once and for all how this file handling mechanism works.

Perhaps the best way of visualizing what is taking place is by means of a simple diagram. Consider Figure - 1

OPEN 2, 8, 3, "#"

Figure - 1

In fact, what Commodore refers to as a 'file#' is actually a 256 byte buffer in RAM, as is the channel#, in this case in the 1541 RAM. The numbers in the 'OPEN' statement are simply your logical numerical labels, and may be chosen to

suit your purposes. In the statement in Figure - 1, we are simply saying "Open a Direct Access path (the '#' symbol) to Device #8, using logical buffer #2 in the C-64, and logical buffer #3 in the disc drive." To place data in these buffers involves commanding the disc drive to READ a TRACK/SECTOR, and this can be achieved by means of the 'U1' command.

Remember that Channel #15 is the path by which commands are given to the disc drive, thus to place the data of TRACK 18/SECTOR 1 into this already defined buffer, we execute:-

```
PRINT #15,commandstring;channel#;device#;track#;sector#
```

or, filling in the blanks;

```
PRINT #15,"U1";3;8;18;1
```

After this command has executed, the contents of T-18/S-1 will be in logical buffer #3 in the 1541 RAM. To fetch the data into the C-64 RAM, we can use either INPUT # or GET #. Since INPUT # will interpret certain characters as delimiters, and, in general, we cannot prevent these characters from being present, we must use GET #, which can accept any character, including NULL.

Thus, to complete the illustration, we would execute:-

```
FOR I=0 TO 255
GET #2,Z$:IF Z$="" THEN Z$=CHR$(0)
DATA$=DATA$+Z$
NEXT I
```

The point to grasp is that whilst we usually make the file# and the channel# the same value, there is no logical reason to do so. Indeed, if we ever get involved with devices such as plotters, then we may well have to come to grips with having a number of files open with channel numbers (or secondary address, which amount to the same thing) that are not the same as the logical file#.

It is also important to grasp that the file#/channel# are actual areas of RAM, and can be referred to directly, if necessary. As shown earlier, there is a fixed relationship between the TASK LIST and its associated buffer. The number of the buffer is not the same as the logical number that you assign, but simply the order in which DOS will assign these buffers as you open more paths to the disc.

The OPEN statement has two other peculiarities that are worth mentioning. The first concerns the use of "file numbers" greater than 127. The 1541 DOS manual indicates (on page 14) that file numbers higher than 127 will cause a LF to be sent after a CR. This is a throw-back to the days when TELETYPES were used as the console device, and a NEWLINE function had to be implemented with the CR+LF

combination. The point to note is that this additional character will only be added at the end of a line of data that is being sent to a sequential or relative file that is being written to the disc, it will not be sent on the serial bus to a device such as the printer. Thus, in general, one should avoid using file numbers greater than 127.

The second peculiarity concerns the 'filestring'. This descriptor is used for a wide range of tasks, and it is not surprising that the range of meanings associated with it is very large. In the case of a Direct Access path, if the filestring descriptor is "#" then it signifies to DOS to "use the next available buffer". If you wish to force DOS to use a particular buffer, then use the form "#n" where 'n' can be from 0 to 4. In this case you will force DOS to choose a specific area in RAM in which to place your data. If this is the first path you have opened to the disc, you have free choice, but if you have already opened a path, then your choice of buffer areas are restricted. If the desired buffer is already in use, DOS will return an error message via Channel #15, "70 NO CHANNEL"

To get round this error, simply start at '0' and increment until no error is returned from the Command/Status Channel.

In the final part of this series, we shall examine the subtleties of using the KERNAL ROM routines to open direct access paths, and will explore the use of the major I/O routines. In addition, we shall explain the operation and use of DISKFIX.

DISKFIX - Part 3

In this, the final part of the DISKFIX saga, I propose to show how to use some of the fundamental KERNAL ROM routines, as well as explaining the use of the DISKFIX program. DISKFIX makes considerable use of the PROMAL JSR function to access the KERNAL I/O routines, and thus you should be familiar with the use of this important function prior to attempting to understand the following explanation. Since this is more than adequately covered in the PROMAL users guide, it will not be replicated here. Table 1 lists the KERNAL ROM routines used by DISKFIX.

NAME	ADDRESS	FUNCTION
CHKIN	$FFC6	Switch INPUT channel to file #.
CHKOUT	$FFC9	Switch OUTPUT channel to file#.
CHROUT	$FFD2	General Character output routine.
CHRIN	$FFCF	General Character input routine.
CLRCHN	$FFCC	Reset I/O switch to KB and SCRN.

Table - 1

At this point, it is appropriate to explain the way in which I/O is controlled in the COMMODORE-64. When the machine is first powered up, the input is expected from the keyboard, whilst output is expected to go to the screen. To alter this source/destination, involves "switching" the input or output to a designated channel. The computer can only input or output data from/to a single source/destination at a time, and in this respect, is analogous to a stereo system. In a stereo system, there is an input switch to select the source of input from phono, tape, radio etc, as well as a loudspeaker switch to send the output the A speakers, the B speakers etc. In the stereo system these input and output switches are operated manually, whilst in the computer, the "switches" are set or reset via software. Switching the input or output is performed by first OPENING a channel, and then using CHKIN to switch the input from the keyboard to the channel, or alternatively, using CHKOUT to switch the output to the channel. To reset the input and output to the keyboard/screen, the CLRCHN routine is used.

Since DISKFIX performs all its I/O via the Error/Status Channel, #15, and since this channel is always open in PROMAL, there is no requirement to open this channel explicitly. Thus, in DISKFIX, it is only necessary to switch the input or output as required to get or send data from/to the disc drive. In order to switch input from the keyboard, the CHKIN routine has to be called with the machine X register set to the logical file number, in this case, #15. To switch output, the CHKOUT routine has to be used, again with the X register set to the logical file

number, #15.

To restore the normal keyboard/screen settings, the CLRCHN routine has to be called. In this case, no registers need to be preset. Note that it is possible to restore the input to the keyboard by calling CHKIN with the X register set to $00, and similarly, the output may be reset to the screen by calling CHKOUT with the X register set to $03. Since CLRCHN performs these chores in a single step, it is usually more convenient to use this routine.

It is important to note that it is advisable to reset the input and output settings after each input or output operation, even if the same channel is to be used again. It was found during the developement of the DISKFIX program that if this was not done, then sequential access to the disc drive would not necessarily behave as expected. Similarly, it was found that the syntax of the commands sent to the disc via the direct use of the KERNAL routines was critical. If you examine the 1541 Users Handbook, you will find that there are alternative forms of many of the direct access commands. For example, the U1 command to read a specific track and sector is shown as having the form (in BASIC. of:-

```
PRINT #15,"U1";2;0;18;0
```

However, if the command is to be sent as a string of characters, then the semi-colon field delimiter ";" is unacceptable. The only acceptable delimiters are the comma, "," or the space. Hence if you wish to send the above command as a string, in BASIC you would have to write:-

```
C$="U1,2,0,18,0"
PRINT #15,C$
```

or,

```
C$="U1 2 0 18 0"
PRINT #15,C$
```

This is also true if you wish to use machine language or the PROMAL JSR function. In this case a "COMMAND-BUFFER" will have to be defined, and the string of characters "U1,2,0,18,0" placed in the buffer with the MOVSTR procedure.

Thus, to emulate the BASIC code, the following PROMAL fragment would need to be executed:-

```
MOVSTR "U1,2,0,18,0",CMDBUF     ;set up command buffer
LENGTH=LENSTR(CMDBUF)           ;get length of command string
POINTER=CMDBUF                  ;get pointer to command string
JSR CHKOUT,0,15                 ;switch output to Channel #15
FOR I=0 TO LENGTH-1
  JSR CHROUT,(POINTER+I)@<      ;output characters
JSR CLRCHN                      ;reset I/O.
```

NB. The code above is not meant to be a stylistic example of the "best" way to achieve the desired action, but rather as an example of the various process that need to be performed and the order of their execution. Obviously, it is possible to economise by combining several operations.

Similarly, it was found that the direct access command "M-R" would mis-behave if the alternate form was used. The usual form of this command is:-

```
"M-R" CHR$(lo-byte) CHR$(hi-byte) CHR$(number of bytes)
```

However, the alternate form:-

```
"M-R:" CHR$(lo-byte) CHR$(hi-byte) CHR$(number of bytes)
```

is sometimes used. The presence of the colon ":" after the command has been found to cause problems - since it is not required, leave it out. As an example, examine the code for the FETCH module to see how direct access commands may be formatted.

Finally, we can now examine the DISKFIX program itself. As you will see from the source file, the actual program is extremely compact:-

```
BEGIN
BORDER=6
SCREEN=0
SWITCH=$17
PUT CLR
BUILDSCREEN
INIT
REPEAT
  SCANKEY
UNTIL KEYCODE=F8
TERMINATE
END
```

The program is built from 24 modules, plus the standard LIBRARY functions. Each of these modules consists of only a few lines of code, and performs a single simple function. For example, the function ADDR contains only a single line of code, whilst the longest module, the procedure WINDOW, contains only 44 actual lines of code representing 12

choices of screen update data, each choice being no more than 4 lines of code. This structure illustrates the modularity of PROMAL programs, and shows clearly how easy it is to construct functionally complex programs from individually simple modules.

When the program is executed, you will be presented with a display screen showing the contents of Track 18 Sector 1. This is the first directory sector, and is usually the first place to start. Notice that the display window at the top of the screen has been optimised to reveal the organisation of the directory. Hence, it is very easy to see at a glance the file names, forward link pointers, file type descriptors, and file length data.

Depressing the "+" key will cause the next sector to be displayed. If the current sector is already the last valid sector, the count will "wrap-around" to the first sector, #0. Similarly, if the "-" key is depressed, the previous sector will be displayed. Again, sector wrap-around will occur if you are already at sector 0, and the highest numbered sector for the particular track where you are located will be displayed.

Depressing the F1 key will cause the sector pointed to by the forward link pointer bytes to be displayed, and the track and sector windows to be updated, as well as the link byte display. If the current sector is the last in the file, depressing the F1 key will cause the END-OF-FILE warning message to be displayed.

Depressing the F2 key allows you to select the track and sector to be displayed. The display window will be highlighted in reverse video, and hitting RETURN will cause the next window to be selected or the entry completed.

The cursor keys behave as expected, and allow the cursor to be positioned anywhere in the display window. Notice that the number displayed in the CSR window represents the position of the cursor in the 256 byte sector. If the F3 key is depressed, the character under the cursor may be edited. Simply enter the decimal number desired, or hit RETURN and enter the desired hexadecimal number. This feature is useful for changing the file type descriptor byte for example to "unscratch" a deleted file. Notice that the cursor does not move after a character update.

Depressing F4 switches you to the EDIT TEXT mode. In this mode alpha-betic characters may be entered, and the cursor will move to the next character. The cursor keys operate normally allowing multiple corrections to be made. Normally, upper case characters are entered in this mode. Depressing the SHIFT key allows the lower case characters to be used. The EDIT TEXT mode is useful for modifying the name of a directory entry for example.

The F5 key is probably the most dangerous key of all. This key allows you to re-write the currently displayed sector data back to the disc. Normally the data would be edited, and then re-written back to its original location.
However, if the F2 key is depressed, it is possible to move the data to another location on the disc. In order to allow an escape, note that a warning prompt is given, and any entry other than F2 or "Y" will cause this mode to abort. After depressing F2, F5 must be re-selected to re-write the data.

Since DISKFIX does not use the DOS, it is possible to write to discs that have had the disc DOS ID changed. Normally, byte 2 of the BAM contains the ASCII "A" character to signify that this is a 1541 disc. If this character is changed then the DOS is unable to write to the disc, and the error message DOS MISMATCH is displayed. Some disc protection schemes deliberately alter this byte to "write-protect" the disc. DISKFIX will ignore this error, and allow this byte to be changed at will and to write to a disc that is otherwise incapable of being written to.

Similarly, since DISKFIX makes no checks of the BAM, it will allow you to read a sector from one disc, to remove the disc from the drive and replace it with another, and to write to the second disc. Be careful! This tool can rescue badly "poisoned" discs, but with only a moments carelessness can also cause considerable damage. Practice on a sacrificial disc before using in anger!

Depressing the F8 key will allow you to exit from DISKFIX. A prompt is displayed asking whether you wish to update the BAM or not. If you have done anything to alter the number or mapping of the used sectors, then you should update. For example, if you have used DISKFIX to recover a scratched program, then the BAM must be updated to prevent the DOS from over-writing the recovered file. The updating is performed by using the DOS "VALIDATE" command, and this will read every file on the disc and allocate sectors in the BAM according to the contents of each file. This process can be quite time consuming, so be prepared to wait for several minutes with a full disc.

If you choose not to update the BAM then DISKFIX will exit after initialising the drive. During the developement process it was found that if the drive was not initialised then a strange illegal error message could be provoked from the 1541. This usually occurred if discs had been swapped and is presumed to occur as a result of an internal inconsistency between the BAM in the 1541 RAM and that on the disc. Initialisation proved to be the complete cure.

Finally, feel free to copy and distribute DISKFIX. I have placed it in the public domain in the hope that it will prove to be a useful and instructive tool.

PROMAL

Public Domain Library
Commodore Disk Number 2

PDC - 002

Systems Management Associates
3325 Executive Drive, P.O. Box 20025
Raleigh, North Carolina 27619

Contents of Commodore PROMAL Public Domain Disk #2

PLEASE NOTE: All public domain disk materials are contributed works. SMA is only serving as a clearing house for these materials as a service to our PROMAL customers. You may copy, use, and further disseminate all Public Domain Disk materials as you see fit. All materials are supplied "as is". SMA does not support these programs in any way. PLEASE DO NOT CALL with questions.

Screen creator by Rev. Mike Cargill

SCREEN.(S,C) = program to create
GET_SCREEN.S = procedure for reading the created screens.
SCREEN.T = documentation

A help screen or the like can be created and stored as a disk file. The displaying program (which you write) can read the screens using the procedure in GET_SCREEN.

Printer control issuer by Julia Christianson

PRINT1.(S,C) = the program
PRINT1.T = documentation

This demo allows various printer control characters to be issued to the printer, one at a time, from the keyboard.

PROMAL source file lister by Garth Ingram

PRINT2.(S,C) = the program
PRINT2_DOC.D = documentation

Prints with page headings. Also, controls can be imbedded in the source code (they look like comments) to eject to a new page and suppress and resume printing.

Document Formatter by David Long

DOCFOR.(S,C) = the program. Also include files DOCFOR1.S, DOCFOR2.S. DOCFOR1.T, DOCFOR2.T, DOCFOR3.T = documentation, in DOCFOR input file format. A printed copy of this document is also included.

The document formatter provides a means of word processing in which an input file is prepared with formatting commands imbedded in the text. The input file is fed into the document formatter, which acts upon the commands and produces an output file that is ready for printing.

Screen creator by W. A. Marsh

CONSTRUCT.(S,C) = the program

Same purpose as previously described screen creator. This one supports color screens, the "price" being that a screen definition requires a larger number of bytes. Consult the source code for usage information.

Graphics routines and demo by Roger Norrod

GRAPHLIB_2.S = an include file having the graphics routines
GRDEMO.(S,C) = demo program (includes GRAPHLIB_2.S)
GRLIB_DOC.S = documentation.

These routines support the high-res screen. Drawing is done primarily by specifying pen moves (draw, erase, complement and "pen up"). ASCII characters can also be generated. A screen can be stored on disk and later recalled. The demo program gives a tour through the features. **NOTE:** this package does not use the PROMAL Graphics Toolbox, nor is it compatible with it or supported in any way by SMA. You may wish to study the PROMAL Graphics Toolbox before committing to a particular graphics package.

Lister that includes time and date stamp by Michael T. Veach

PR.(S,C) = lister program
SET_TIME.(S,C) = sets date and time
PPDL.T = documentation

First, a timer is initialized using SET_TIME (and also a date entered). Thereafter when PR is run, the date and time is printed at the top of each page.

C64-to-Tandy PC2 data exchange program by Steve Vermeulen

PC2.(S,C) = the program

Files can be passed between these two computers over an RS-232 bus. The Tandy PC2 is identical to the Sharp PC-1500.

"Dumb terminal" emulation routine and demo by Steve Vermeulen

DUMBTERM.S = the emulation routine, an include file
DEMOTERM.(S,C) = demo using DUMBTERM.S

DUMBTERM.S is an address-independent machine language routine coded as a PROMAL DATA statement. DEMOTERM.C has been used to transfer data from a C64 to a Harris 800 computer.

KOALA touchpad support by Erik Vigmostad

KOALA.(S,C) = the program

A design on the screen is constructed under control of the KOALA touchpad.

Counter of word occurrances in a file by Erik Vigmostad

COUNT.(S,C) = the program

The input to this program is a file containing words, such as a text or PROMAL source code. The output is a list of each different word found along with the number of times it occurred.

File Lister for RS-232 printer by Erik Vigmostad

PRINT3.(S,C) = the program.
RS_232.S = include file for PRINT3.S.

A convenience for PROMAL users with RS-232 printers. Without a program such as this one, it is necessary to exit PROMAL and use a BASIC program to do file printing.

DOCFOR - A PROMAL Document Formatter
by David Long

Notes from SMA

This document formatter program is a PROMAL implementation of a design similiar to one given in Software Tools in Pascal by Kernighan and Plauger, published by Addison-Wesley, 1981.

DOCFOR is a public domain program submitted by a user. You can freely use and copy it. SMA serves only as a clearing house for such programs, and SMA in no way stands behind their correctness, nor does SMA provide support for them.

You may need to modify the printer part for your particular printer before doing underlining. See instructions in DOCFOR.S. The presently implemented method of printing underlines consists of printing a BS and '_' sequence after each character to be underlined. This works on many printers.

The files associated with this program (found on Public Domain Disk #2) are.

DOCFOR.S	Source file for main program. It uses include files DOCFOR1.S and DOCFOR2.S
DOCFOR.C	Executable version of the program.
DOCFOR1.T DOCFOR2.T DOCFOR3.T	Documentation. These three files, when fed into DOCFOR, produce a document identical to the one you are looking at. They illustrate the use of DOCFOR.

INTRODUCTION

This program offers a powerful and well established method of word processing that begins with a manually created file containing text intermixed with formatting commands. The manually created file, which makes no pretense of looking like the final document, is fed into the document formatter program, which acts upon the commands and produces an output file that is in final form for printing.

A major advantage of this approach is simplicity. The input file can be created on just about any editor, the PROMAL EDITor being an excellent candidate. Formatting commands are short sequences of "ordinary" characters that are always visible. There is nothing hidden from view. The formatting commands perform the following general functions:

* Packing words together to form complete lines
* Producing text with flush left and right margins
* Centering text on a line
* Creating headers and footers, which may include a page number.

USING THE FORMATTER

After creating an input file, start the formatter from the EXECUTIVE by

typing

```
DOCFOR input_file [output_file]
```

where input_file is the name of the file to be formatted and output_file is the name of the file to receive the formatted text. If input_file has no extension, a default extension of .T is assumed. If output_file is not specified, it is taken to be input_file with a .F extension. If output_file is specified but has no extension, the .F extension is assumed.

For example,

```
DOCFOR ESSAY
```

formats the file ESSAY.T and sends the output to ESSAY.F. The command

```
DOCFOR REPORT MEMO.T
```

processes REPORT.T and creates MEMO.T. Frequently you will wish to output directly to the printer. That can be done by specifying output to "P", which in PROMAL is the printer device. For example,

```
FORMAT TEXT.S P
```

formats TEXT.S and prints the result.

FORMATTING COMMANDS

The formatting commands are placed amongst the text in the input file to control how the text will look in the final document. Each command is immediately preceeded by a period and starts at the beginning of a line that contains nothing but the command and its operands. Any line beginning with a period is interpreted as a command.

Many commands take an argument of some form. An argument is any text following a command and seperated from it by at least one space. There are three types of arguments: absolute numbers, relative numbers, and text. Absolute numbers are denoted by an unsigned integer, and are used to set parameters to a specific value. Relative numbers are integers preceded by a + or a -, and are used to change a parameter relative to its current value. If an argument is not in either of these classes, it is considered to be text.

For example, the command

```
.RM 65
```

sets the right margin to a value of 65. The command

```
.TM-1
```

repositions the top margin one line above what it formerly was. The command

```
.EF /January Report/ZZZ, Inc./New Products/
```

defines a page footer for even-numbered pages.

PAGE LAYOUT

Vertically, each page has seven regions: the header margin, the header, the top margin, the page body, the bottom margin, the footer, and the footer margin. The header margin separates the header from the top of the page. Similarly, the footer margin separates the footer from the bottom of the page. The top and bottom margins separate the page body from the header and footer.

Horizontally, all text is indented to the left margin, and all text ends before the right margin.

Here is an illustration:

```
Left margin                    Right margin
|                                        |
V                                        V
----------------------------------------  <- Top of page
                                          <- Header margin

Page 14      Learning PROMAL     SMA, Inc.  <- Header
                                          <- Top margin
The CHOOSE statment is a convenient way   <- Page body

                      ...

When the number of iterations is known
                                          <- Bottom margin
       (C) 1985 SMA, Inc.                 <- Footer
                                          <- Footer margin

----------------------------------------  <- Bottom of page
```

SUMMARY OF COMMANDS

.FI	Fill (default)
.NF	Stop filling
.JU	Justify (default)
.NJ	Stop justifying
.CE	Center
.NC	Stop centering (default)
.UL	Underline
.NU	Stop underlining (default)
.IN n	Indent (set left margin to) n (default 0)
.TI n	Indent an additional n on next line (default 5)
.RM n	Set right margin to n (default 60)
.TM n	Set top margin to n (default 2)
.HM n	Set header margin to n (default 2)
.BM n	Set bottom margin to n (default 2)

```
.FM n            Set footer margin to n (default 2)
.PL n            Set page length to n (default 66)
.LS n            Set line spacing to n (default 1, single space)
.SP n            Skip n lines (default 1)
.OH /lt/mt/rt/   Set odd header to text
.OF /lt/mt/rt/   Set odd footer to text
.EH /lt/mt/rt/   Set even header to text
.EF /lt/mt/rt/   Set even footer to text
.BR              Break (start new line)
.BP n            Begin page number n (default +1)
.CP n            New page if less than n lines left (default 1)
.LI n            Emit literal character n (no default)
```

DETAILED DESCRIPTION OF COMMANDS

.FI - Fill Text

The fill text command causes the formatter to collect as many words as possible on each output line, i.e., words will be taken from the input lines as needed to form an output line which is as long as possible (without exceeding the margins). Filling defaults to "on".

For example,

```
.FI
The quick brown fox
jumped over
the lazy dogs.
A bird in the hand is worth two
in the bush.
```

produces

> The quick brown fox jumped over the lazy dogs. A bird in the hand is worth two in the bush.

.NF - Stop Filling

The stop filling command causes the formatter to print each output line with only the words on the corresponding input line. No words will be moved off of or onto the line to make it fit the margins.

For example,

```
.FI
The quick brown fox
jumped over
the lazy dogs.
.NF
A bird in the hand is worth two
in the bush.
```

produces

```
The quick brown fox jumped over the lazy dogs.
A bird in the hand is worth two
in the bush.
```

.JU - Justify Text

The justify text command causes the formatter to insert blanks in a line in order to make a flush right margin. Justification defaults to on.

For example,

```
.FI
.JU
The quick brown fox
jumped over the
lazy dogs.  A bird in the
hand is worth two
in the bush.
Now is the time for all
good men to come to the aid of
their country.
We the people, in order to form
a more perfect union, establish
justice, insaure domestic tranquility,
provide for the common defence, promote the
general welfare, and secure the blessings of
liberty, to ourselves and our posterity, do
ordain and establish this constitution for the
United States of America.
```

produces

```
The quick brown fox jumped over the lazy dogs.  A bird in the
hand is worth two in  the bush.  Now is the time for all good
men to come to the  aid of their country.  We the people, in
order to form a more perfect union, establish justice, insure
domestic tranquility, provide for the common defense, promote
the general welfare,  and secure our posterity, do ordain and
establish this constitution  for the United States of America.
```

.NJ - Stop Justifying

When justication is turned off, the formatter will not insert spaces in order to make a line flush with the right margin. For example,

```
.FI
.NJ
The quick brown fox
jumped over the
lazy dogs.  A bird in the
hand is worth two
```

```
in the bush.
Now is the time for all
good men to come to the aid of
their country.
We the people, in order to form
a more perfect union, establish
justice, insure domestic tranquility,
provide for the common defence, promote the
general welfare, and secure the blessings of
liberty, to ourselves and our posterity, do
ordain and establish this constitution for the
United States of America.
```

produces

> The quick brown fox jumped over the lazy dogs. A bird in the hand is worth two in the bush. Now is the time for all good men to come to the aid of their country. We the people, in order to form a more perfect union, establish justice, insure domestic tranquility, provide for the common defense, promote the general welfare, and secure our posterity, do ordain and establish this constitution for the United States of America.

.CE - Center Text

The center text command causes the following lines to be centered on the page. During centering, justification and filling are turned off. Centering defaults to off.

For example,

```
.CE
January Report
ZZZ, Inc.
```

produces

January Report
ZZZ, Inc.

.NC - Stop Centering

The stop centering command turns off centering of text. Previous justification and filling settings are restored.

For example,

```
.CE
January Report
ZZZ, Inc.

.NC
During the fourth quarter of last year,
sales were at a record high.
```

produces

January Report
ZZZ Inc.

During the fourth quarter of last year, sales were at a record high.

.UL - Underline Text

The underline text command causes the following text (but not headers or footers) to be underlined. Underlining is accomplished by printing an underscore, a backspace, and then the character to be underlined. Underlining defaults to off.

For example,

```
.UL
This text is underlined.
```

produces

This text is underlined.

.NU - Stop Underlining

The stop underlining command turns off underlining. For example,

```
.FI
.UL
This text is underlined,
.NU
but this isn't.
```

produces

This text is underlined, but this isn't.

.IN n - Indent, or Set Left Margin

The indent command sets the left margin for all text. If n is an absolute number, the margin will be set to that value. If n is relative, the margin will be moved by n from its present position (this is useful for indenting quotes). The margin defaults to 0 (no spaces before text).

For example,

```
Shakespeare once wrote:
.IN +5
To be or not to be, that is the question.
```

produces

```
Shakespeare once wrote:
     To be or not to be, that is the question.
```

.TI n - Temporary Indent

The temporary indent command indents the next line an extra n spaces. Since the temporary indent value is set to 0 after each line, relative and absolute numbers are equivalent. This command is usually used to begin a new paragraph. The default value of n is 5.

For example,

```
.TI
Now is the time for all good
men to come to the aid of their country.
The quick brown fox jumped over
the lazy dogs.
```

will produce

```
     Now is the time for all good men to come to the aid of
their country.  The quick brown fox jumped over the lazy dogs.
```

Here is an example of .TI in conjunction with .IN:

```
.IN +5
.TI -5
Once upon a time, there was a girl
named Mary, who had a very small and very
white lamb.
```

This produces:

```
Once upon a time, there was a girl named Mary, who had a
     very small and very white lamb.
```

.RM n - Set Right Margin

The set right margin command sets the right margin for all text. The number n may be absolute or relative, and defaults to 60.

For example,

```
.FI
.RM 40
Now is the time for all good men to come to the aid of
their country.
```

produces

```
Now is the time for all good
men to come to the aid of
```

their country.

.TM n - Set Top Margin

The set top margin command sets the top margin; i.e., the margin between the header and the page body. The number n may be absolute or relative and defaults to 2.

For example,

```
.TM 4
```

produces 4 lines between the header and page body starting with the next page.

.HM n - Set Header Margin

The set header margin command sets the margin between the top of the page and the header. The number n may be absolute or relative and defaults to 2.

For example,

```
.HM +2
```

produces 2 additional lines between the top of the page and the header starting with the next page.

.BM n - Set Bottom Margin

The set bottom margin command sets the margin between the page body and the footer. The number n may be absolute or relative and defaults to 2.

For example,

```
.BM
```

produces 2 lines (the default) between the bottom of the page body and the footer starting with the end of this page.

.FM n - Set Footer Margin

The set footer margin command sets the margin between the footer and the end of the page. The number n may be absolute or relative and defaults to 2.

For example,

```
.FM 3
```

produces 3 lines between the footer and the bottom of the page starting with the end of this page.

.PL n - Set Page Length

The set page length command sets the total number of lines per page. The number of actual text lines printed will be determined by this number and the margin settings. The number n may be absolute or relative and defaults to 66.

For example,

```
.PL 84
```

sets the length of the page to 84 lines (legal size).

.LS n - Set Line Spacing

The set line spacing command sets the number of lines to skip between text lines. The number n may be absolute or relative and defaults to 1 (single spacing).

For example,

```
.NF
.LS 1
These lines are
single spaced.
.LS 2
These are double
spaced.
```

produces

```
These lines are
single spaced.
These are double

spaced.
```

.SP n - Skip Lines

The skip lines command produces blank lines. It does not produce blank lines past the end of a page. For example, if there are 2 lines left in the page body and ".SP 3" is executed, only 2 blank lines are produced. To produce blank lines at the top of a page, use the begin page (.BP) command, then the skip lines command. Then number n may be absolute or relative and defaults to 1.
For example,

```
.NF
.LS 1
Now is the time for all good men to come
.SP
to the aid of their country.
```

produces

Now is the time or all good men to come

to the aid of their country.

.OH /lt/mt/rt/ - Set Odd Header

The set odd header command sets the text which appears at the top of odd numbered pages to the given text. The header is a single line, which is given in the form of three fields, shown above as "lt", "mt" and "rt", meaning "left text", "middle text" and "right text". The "lt" field is flushed left in the header, the "mt" field is centered, and the "rt" field is flushed right.

In the above symbolism, the fields are delimited by "/". In actuality, any nonblank character except "#" can be used as the delimiter as long as it does not appear in the text fields.

It is your responsibility to see that the three fields are not so long as to fall on top of each other.

Whenever a "#" appears in a text field, it is replaced by the current page number.

For example,

.OH "January Report"ZZZ, Inc."Page #"

produces the following header:

January Report ZZZ, Inc. Page 32

.OF /lt/mt/rt/ - Set Odd Footer

This command is identical to the preceding "set odd header" command except that a footer is generated at the bottom of each subsequent odd-numbered page.

.EH /lt/mt/rt/ - Set Even Header

This command is identical to the "set odd header" command except that the header is generated on each subsequent even numbered page.

.EF /lt/mt/rt/ - Set Even Footer

This command is identical to the "set odd footer" command except that the footer is generated on each subsequent even numbered page.

.BR - Break

The break command is used only in fill mode. It causes the output line which is being built to be printed immediately with no further filling and no justific tion. Many other commands, such as temporary indent, automatically cause a break.

For example,

```
.FI
.JU
Now is the time for all good men
.BR
to come to the aid of their country.
```

produces

```
Now is the time for all good men
to come to the aid of their country.
```

.BP n - Begin Page

The begin page command causes the formatter to begin a new page of output. It has no effect at the top or bottom of a page. If n is specified, it sets the number for the next page to be printed. The number n may be absolute or relative, and defaults to +1.

For example,

```
.BP 10
```

begins a new page (if currently in a page body) and sets the page number to 10.

.CP n - Conditional Page

The conditional page command causes a new page if there are less than n lines remaining on the current page. The number n may be absolute or relative and defaults to 1. It will not start a new page if the formatter is currently at the bottom of a page.

For example,

```
.CP 15
```

starts a new page if the body of the current page has less than 15 lines left.

.LI n - Literal Character

The literal character command embeds a literal character in the output stream. This is typically used to send a special command to the print for something such as font selection. The number n should be an absolute number between 0 and 255, and ASCII character number n will be output. Note that the formatter does not take the effect of any literal

characters into account, so a page break could result in a strange header or footer. If you are in the middle of a page and need to skip the footer before emitting the character, use a .BP command. If you are at the top of the page and need to skip the header, use a

For example,

```
.LI 27
.LI 52
Slanted letters
.LI 27
.LI 53
```

prints "Slanted letters" in italics on an Epson MX80, and

```
.LI 27
.LI 69
Emphasized letters
.LI 27
.LI 70
```

prints "Emphasized letters" in emphasized mode on an MX80.

SAMPLE DOCUMENT, before and after formatting

The following is an example of a typical document designed for use with the formatter:

```
.EF ""-#-""
.OF ""-#-""
.NF
.NJ
Twobit Computer Company
4321 Somewhere Drive
Nowheresville, NC 27610
.SP 2
.FI
Dear sirs:
.SP
.JU
.TI
Recently, I purchased a Twobit widget enchancer and
kludger (TWEAKer) for use with my Machturbo Hypersonic
69000-e computer.  After using the product for several
weeks, I found that it was incompatible with my
Tangledfingers Ultracomplex File Lister with Unbelievable
Captions (TUFLUC).
.SP
.TI
When I attempted to use the TWEAKer with my TUFLUC, I
found that the screen emitted a primordial scream, and
that the disk drives produced large amounts of fire and
red smoke.  Just in case this was a fluke, I repeated the
process and obtained the same result.  I suspect that
the problem is a result of my custom written Dumb Operating
System (DOS).
.SP
.TI
If you have heard of a similar problem and have a fix for
it, I would appreciate receiving it (I really like the TWEAKer).
If you do not know of a fix, I would appreciate a refund,
in addition to $5.67 to cover the cost of a black tie (burned
by the disk drives).  I realize that in the licence
agreement for the TWEAKer, it says that I am your slave
for life, so I can only appeal to your humanity.
.SP 5
.NF
.NJ
.IN +48
Joe Customer
.IN -48
```

Formatting the above results in the following:

Twobit Computer Company
4321 Somewhere Drive
Nowheresville, NC 27610

Dear sirs:

Recently, I purchased a Twobit widget enhancer and kludger (TWEAKer) for use with my Machturbo Hypersonic 69000-e computer. After using the product for several weeks, I found that it was incompatible with my Tangledfingers Ultracomplex File Lister with Unbelievable Captions (TUFLUC).

When I attempted to use the TWEAKer with my TUFLUC, I found that the screen emitted a primordial scream, and that the disk drives produced large amounts of fire and red smoke. Just in case this was a fluke, I repeated the process and obtained the same result. I suspect that the problem is a result of my custom written Dumb Operating System (DOS).

If you have heard of a similar problem and have a fix for it, I would appreciate receiving it (I really like the TWEAKer). If you do not know of a fix, I would appreciate a refund, in addition to $5.67 to cover the cost of a black tie (burned by the disk drives). I realize that in the licence agreement for the TWEAKer, it says that I am your slave for life, so I can only appeal to your humanity.

Joe Customer

(blank lines to bottom of page)