Part 2 / Software

1/Disk Organization

TRSDOS Version 6 can be used with 5¼" single-sided floppy diskettes and with hard disk. Floppy diskettes can be either single- or double-density. See the charts below for the number of sectors per track, number of cylinders, and so on for each type of disk. (Sectors and cylinders are numbered starting with 0.)

Single-Density Floppy Diskette

Bytes per Sector	Sectors per Granule	Sectors per Track*	Granules per Track	Tracks per Cylinder	Cylinders per Drive	Total Bytes
256						256
	5					1,280
		(10)	2			2,560
				1		2,560
					40	102,400
256	5	(10)	2	1	40	102,400 (100K)**

Double-Density Floppy Diskette

Bytes per Sector	Sectors per Granule	Sectors per Track*	Granules per Track	Tracks per Cylinder	Cylinders per Drive	Total Bytes
256						256
	6					1,536
		(18)	3			4,608
				1		4,608
					40	184,320
256	6	(18)	3	1	40	184,320 (180K)**

*The number of sectors per track is not included in the calculation because it is equal to the number of sectors per granule times the number of granules per track. ($5 \times 2 = 10$ for single density, $6 \times 3 = 18$ for double density, and $16 \times 2 = 32$ for hard disk.)

**Note that this figure is the total amount of space in the given format. Keep in mind that an entire cylinder is used for the directory and at least one granule is used for the bootstrap code. This leaves 96.25K available for use on a single-density data disk and 174K on a double-density data disk.

5" 5-Meg Hard Disk

Note: Because of continual advancements in hard disk technology, the number of tracks and the number of tracks per cylinder may change. Therfore, any information that comes with your hard disk drive(s) supersedes the information in the table below.

Bytes per Sector	Sectors per Granule	Sectors per Track*	Granules per Track	Tracks per Cylinder	Cylinders per Drive	Total Bytes
256	----------	----------	----------	----------	----------	256
	16	----------	----------	----------	----------	4,096
		(32)	2	----------	----------	8,192
				4	----------	32,768
					153	5,013,504
256	16	(32)	2	4	153	5,013,504 (4,896K)

*The number of sectors per track is not included in the calculation because it is equal to the number of sectors per granule times the number of granules per track. ($5 \times 2 = 10$ for single density, $6 \times 3 = 18$ for double density, and $16 \times 2 = 32$ for hard disk.)

Disk Space Available to the User

One granule on cylinder 0 of each disk is reserved for the system. It contains information about where the directory is located on that disk. If the disk contains an operating system, then all of cylinder 0 is reserved. This area contains information used to load TRSDOS when you press the reset button.

One complete cylinder is reserved for the directory, the granule allocation table (GAT), and the hash index table (HIT). (On single-sided diskettes, one cylinder is the same as one track.) The number of this cylinder varies, depending on the size and type of disk. Also, if any portion of the cylinder normally used for the directory is flawed, TRSDOS uses another cylinder for the directory. You can find out where the FORMAT utility has placed the directory by using the Free *:drive* command.

On hard disks, an additional cylinder (cylinder 1) is reserved for use in case your disk drive requires service. This provides an area for the technician to write on the disk without harming any data. (If you bring your hard disk in for service, you should try to back up the contents of the disk first, just to be safe.)

Unit of Allocation

The smallest unit of disk space that the system can allocate to a file is a granule. A granule is made up of a set of sectors that are adjacent to one another on the disk. The number of sectors in a granule depends on the type and size of the disk. See the charts on the previous two pages for some typical sizes.

Methods of File Allocation

TRSDOS provides two ways to allocate disk space for files: dynamic allocation and pre-allocation.

Dynamic Allocation

With dynamic allocation, TRSDOS allocates granules only at the time of write. For example, when a file is first opened for output, no space is allocated. The first allocation of space is done at the first write. Additional space is added as required by further writes.

With dynamically allocated files, unused granules are de-allocated (recovered) when the file is closed.

Unless you execute the CREATE system command, TRSDOS uses dynamic allocation.

Pre-Allocation

With pre-allocation, the file is allocated a specified number of granules when it is created. Pre-allocated files can be created only by the system command CREATE. (See the *Disk System Owner's Manual* for more information on CREATE.)

TRSDOS automatically extends a pre-allocated file as needed. However, it does not de-allocate unused granules when a pre-allocated file is closed. To reduce the size of a pre-allocated file, you must copy it to a dynamically allocated file. The COPY (CLONE = N) system command does this automatically.

Files that have been pre-allocated have a 'C' by their names in a directory listing.

Record Length

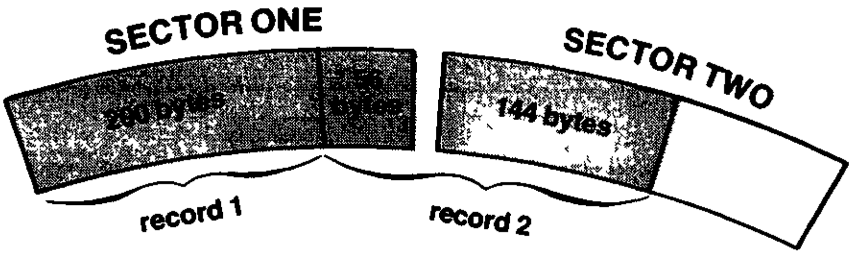
TRSDOS transfers data to and from disks one sector at a time. These sectors are 256-byte blocks, and are also called the system's "physical" records.

You deal with records that are 256 bytes in length or smaller, depending on what size record you want to work with. These are known as "logical" records.

You set the size of the logical records in a file when you open the file for the first time. The size is the number of bytes to be kept in each record. There may be from 1 to 256 bytes per logical record.

The operating system automatically accumulates your logical records and stores them in physical records. Since physical records are always 256 bytes in length, there may be one or more logical records stored in each physical record. When the records are read back from disk, the system automatically returns one logical record at a time. These actions are known as "blocking" and "de-blocking," or "spanning."

For example, if the logical record length is 200, sectors 1 and 2 look like this:



Since they are completely handled by the operating system, you do not need to concern yourself with physical records, sectors, granules, tracks, and so on. This is to your benefit, as the number of sectors per granule varies from disk to disk. Also, physical record lengths may change in future versions of TRSDOS, but the concept of logical records will not.

Note: All files are fixed-length record files with TRSDOS Version 6.

Record Processing Capabilities

TRSDOS allows both direct and sequential file access.

Direct access (sometimes called "random access") lets you process records in any sequence you specify.

Sequential access allows you to process records in sequence: record n, n + 1, n + 2, and so on. With sequential access, you do not specify a record number. Instead, TRSDOS accesses the record that follows the last record processed, starting with record 0.

With sequential access files, use the @READ supervisor call to read the next record, and the @WRITE or @VER supervisor call to write the next record. (When the file is first opened, processing starts at record 0. You can use @PEOF to position to the end of file.)

To read or write to a direct access file, use the @POSN supervisor call to position to a specified record. Then use @READ, @WRITE, or @VER as desired. Once @POSN has been used, the End of File (EOF) marker will not move, unless the file is extended by writing past the current EOF position.

Record Numbers

Using direct (random) access, you can access up to 65,536 records. Record numbers start at 0 and go to 65535.

Using a file sequentially, you can access up to 16,777,216 bytes. To calculate the number of records you can access sequentially, use the formula:

16,777,216 ÷ logical record length = number of sequential records allowed

Below are some examples.

If the LRL = 256, then:
16,777,216 ÷ 256 = 65,536 records

If the LRL = 128, then:
16,777,216 ÷ 128 = 131,072 records

If the LRL = 50, then:
16,777,216 ÷ 50 = 335,544 records

If the LRL = 1, then:
16,777,216 ÷ 1 = 16,777,216 records

3/TRSDOS File Descriptions

This section describes four types of files found on your TRSDOS master diskette (system files, utilities, driver programs, and filter programs) and explains their functions. It also describes how to construct a minimum system disk for running applications packages.

System Files (/SYS)

TRSDOS Version 6 would occupy considerable memory space if all of it were resident in memory at any one time. To minimize the amount of memory reserved for system use, TRSDOS uses overlays.

Using an overlay-driven system involves some compromise. While a user's application is in progress, different overlays may need to be loaded to perform certain activities requested of the system. This could cause the system to run slightly slower than a system which has more of its file access routines always resident in memory.

The use of overlays also requires that a SYSTEM disk usually be available in Drive 0 (the system drive). Since the disk containing the operating system and its utilities leaves little space available to the user, you may want to remove certain parts of the system software not needed while a particular application is running. You may in fact discover that your day-to-day operations need only a minimal TRSDOS configuration. The greater the number of system functions unnecessary for your application, the more space you can have available for a "working" system disk. Use the PURGE or REMOVE library command to eliminate unneeded system files from the disk.

The following paragraphs describe the functions performed by each system overlay. (In the display produced by the DIR (SYS) library command, the system overlays are identified by the file extension /SYS.)

Note: Two system files are put on the disk during formatting. They are DIR/SYS and BOOT/SYS. These files should *never* be copied from one disk to another or REMOVEd. TRSDOS automatically updates any information necessary when performing a backup.

SYS0/SYS

This is not an overlay. It contains the resident part of the operating system (SYSRES). It is also needed to dynamically allocate file space used when writing files. Any disk used for booting the system *must* contain SYS0. It can be purged from disks not used for booting.

SYS1/SYS

This overlay contains the TRSDOS command interpreter and the routines for processing the @CMNDI, @CMNDR, @FEXT, @FSPEC, and @PARAM system vectors. This overlay must be available on all SYSTEM disks.

SYS2/SYS

This overlay is used for opening or initializing disk files and logical devices. It also contains routines for processing the @CKDRV, @GTDCB, and @RENAM system vectors, and routines for hashing file specifications and passwords. This overlay must be available on all SYSTEM disks.

SYS3/SYS

This overlay contains all of the system routines needed to close files and logical devices. It also contains the routines needed to service the @FNAME system vector. This overlay must not be removed from the disk.

SYS4/SYS

This overlay contains the system error dictionary. It is needed to issue such messages as "File not found," "Directory read error," etc. If you decide to remove this overlay from your working SYSTEM disk, all system errors will produce the error message "SYS ERROR." It is recommended that you not remove this overlay, especially since it occupies only one granule of space.

SYS5/SYS

This is the "ghost" debugger. It is needed if you intend to test out machine language application software by using the TRSDOS DEBUG library command. If your operation will not require this debugging tool, you may purge this overlay.

SYS6/SYS

This overlay contains all of the routines necessary to service the library commands identified as "Library A" by the LIB command. This represents the primary library functions. Only very limited use can be made of TRSDOS if this overlay is removed from your working SYSTEM disk.

SYS7/SYS

This overlay contains all of the routines necessary to service the library commands identified as "Library B" by the LIB command. A great deal of use can be made of TRSDOS even without this overlay. It performs specialized functions that may not be needed in the operation of specific applications. You can purge this overlay if you decide it is not needed on a working SYSTEM disk.

SYS8/SYS

This overlay contains all of the routines necessary to service the library commands identified as "Library C" by the LIB command. A great deal of use can be made of TRSDOS even without this overlay. It performs specialized functions that may not be needed in the operation of specific applications. You can purge this overlay if you decide it is not needed on a working SYSTEM disk.

SYS9/SYS

This overlay contains the routines necessary to service the extended DEBUG commands available after a DEBUG (EXT) is performed. This overlay may be purged if you will not need the extended DEBUG commands while running your application. If you remove SYS5/SYS, then you may as well remove SYS9/SYS, as it would serve no useful purpose.

SYS10/SYS

This system overlay contains the procedures necessary to service the request to remove a file. It should remain on your working SYSTEM disks.

SYS11/SYS

This overlay contains all of the procedures necessary to perform the Job Control Language execution phase. You may remove this overlay from your working disks if you do not intend to execute any JCL functions. If SYS6/SYS (which contains the DO command) has been removed, keeping this overlay would serve no purpose.

SYS12/SYS

This system overlay contains the routines that service the @DODIR, @GTMOD, and @RAMDIR system vectors. It should remain on your disks.

SYS13/SYS

This overlay is reserved for future system use. It contains no code and takes up no space on the disk. You may remove this overlay if you wish to free up its directory slot.

- SYS2 must be on the system disk if a configuration file is to be loaded.
- SYS11 must be present only if any JCL files will be used.
- All three libraries (SYS files 6, 7, and 8) may be purged if no library command will be used.
- SYS5 and SYS9 may be purged if the system DEBUG package is not needed.
- SYS0 may be removed from any disk not used for booting.
- SYS11 (the JCL processor) and SYS6 (containing the DO library command) must both be on the disk if the DO command is to be used. Also, if you remove SYS6, you may as well remove SYS11.
- SYS13 may be removed if you have not implemented an ECI, an IEP file, or if you do not intend to use them.

The presence of any utility, driver, or filter program is dependent upon your individual needs. You can save most of the TRSDOS features in a configuration file using the SYSTEM (SYSGEN) command, so the driver and filter programs will not be needed in run time applications. If you intend to use the HELP utility, your disk must contain the DOS/HLP file.

The owner (update) passwords for TRSDOS files are as follows:

File Type	Extension	Owner Password
System files	(/SYS)	LSIDOS
Filter files	(/FLT)	FILTER
Driver files	(/DVR)	DRIVER
Utility files	(/CMD)	UTILITY
BASIC		BASIC
BASIC overlays	(/OV$)	BASIC
CONFIG/SYS		CCC
Drive Code Table Initializer	(/DCT)	UTILITY

Device Control Block (DCB)

The Device Control Block (DCB) is an area of memory that contains information used to interface the operating system with various logical devices. These devices include the keyboard (*KI), the video display (*DO), a printer (*PR), a communications line (*CL), and other devices that you may define.

The following information describes each assigned DCB byte.

DCB+0 (TYPE Byte)

Bit 7 — If set to "1," the Device Control Block is actually a File Control Block (FCB) with the file open. Since DCBs and FCBs are similar, and devices may be routed to files, a "device" with this bit set indicates a routing to a file.

Bit 6 — If set to "1," the device defined by the DCB is filtered or is a device filter.

Bit 5 — If set to "1," the device defined by the DCB is linked.

Bit 4 — If set to "1," the device defined by the DCB is routed.

Bit 3 — If set to "1," the device defined by the DCB is a NIL device. Any output directed to the device is discarded. For any input request, the character returned is a null (ASCII value 0).

Bit 2 — If set to "1," the device defined by the DCB can handle requests generated by the @CTL supervisor call. See the section on Supervisor Calls for more information.

Bit 1 — If set to "1," the device defined by the DCB can handle output requests which normally come from the @PUT supervisor call.

Bit 0 — If set to "1," the device defined by the DCB can handle requests for input which normally come from the @GET supervisor call.

DCB+1 and DCB+2

Contain the address of the driver routine that supports the hardware assigned to this DCB. (In the case of a routed or linked device, the vector may point to another DCB.)

DCB+3 through DCB+5

Reserved for system use.

DCB+6 and DCB+7

These locations normally contain the two alphabetic characters of the devspec. The system uses the devspec as a reference in searching the device control block tables.

Memory Header

Modules that TRSDOS loads into memory (filters, drivers, and other memory modules such as a SPOOL buffer or the extended DEBUG code) are identified by a standard front-end header:

```
BEGIN:  JR   START          ;Go to actual code
                            ;beginning
        DEFW END-1          ;Contains the highest byte
                            ;of memory
                            ;used by the module
        DEFB 10             ;Length of name, 1-15
                            ;characters;
                            ;bits 4-7 reserved for
                            ;system use
        DEFM 'NAMESTRING'   ;Up to 15 alphanumeric
                            ;characters, with the first
                            ;character A-Z. This should
                            ;be a unique name to
                            ;positively identify the
                            ;module.
MODDCB: DEFW $-$            ;DCB pointing to this
                            ;module (if applicable)
        DEFW 0              ;Spare system pointer _
                            ;RESERVED
;
;       Any additional data storage goes here
;
START:  Start of actual program code

END:    EQU $
```

As explained under the @GTMOD SVC in the "Supervisor Call" section, the location of a specific header can be found provided all modules that are put into memory use this header structure. You can locate the data area for a module by using @GTMOD to find the start of the header and then indexing in to the data area.

Drive Code Table (DCT)

TRSDOS uses a Drive Code Table (DCT) to interface the operating system with specific disk driver routines. Note especially the fields that specify the allocation scheme for a given drive. This data is essential in the allocation and accessibility of file records.

The DCT contains eight 10-byte positions — one for each logical drive designated 0-7. TRSDOS supports a standard configuration of two-floppy drives. You may have up to four floppy drives. This is the default initialization when TRSDOS is loaded.

Here is the Drive Code Table layout:

DCT+0

This is the first byte of a 3-byte vector to the disk I/O driver routines. This byte is normally X'C3.' If the drive is disabled or has not been configured (see the SYSTEM command in the *Disk System Owner's Manual*), this byte is a RET instruction (X'C9').

DCT+1 and DCT+2

Contain the entry address of the routines that drive the physical hardware.

DCT+3

Contains a series of flags for drive specifications.

Bit 7—Set to "1" if the drive is software write protected, "0" if it is not. (See the SYSTEM command in the *Disk System Owner's Manual.*)

Bit 6—Set to "1" for DDEN (double density), or "0" for SDEN (single density).

Bit 5—Set to "1" if the drive is an 8" drive. Set to "0" if it is a 5¼" drive.

Bit 4—A "1" causes the selection of the disk's second side. The first side is selected if this bit is "0." This bit value matches the side indicator bit in the sector header written by the Floppy Disk Controller (FDC).

Bit 3—A "1" indicates a hard drive (Winchester). A "0" denotes a floppy drive (5¼" or 8").

Bit 2—Indicates the time delay between selection of a 5¼" drive and the first poll of the status register. A "1" value indicates 0.5 second and a "0" indicates 1.0 second. See the SYSTEM command in the *Disk System Owner's Manual* for more details.

If the drive is a hard drive, this bit indicates either a fixed or removable disk: "1" = fixed, "0" = removable.

Bits 1 and 0—Contain the step rate specification for the Floppy Disk Controller. (See the SYSTEM command in the *Disk System Owner's Manual.*) In the case of a hard drive, this field may indicate the drive address (0-3).

DCT+4

Contains additional drive specifications.

Bit 7— (Version 6.2 only) If "1", no @CKDRV is done when accessing the drive. If an application opens several files on a drive, this bit can be set to speed I/O on that drive after the first successful open is performed.

In versions prior to TRSDOS 6.2, this bit is reserved for future use. In order to maintain compatibility with future releases of TRSDOS, do not use this bit.

Bit 6 — If "1", the controller is capable of double-density mode.

Bit 5 — "1" indicates that this is a 2-sided floppy diskette; "0" indicates a 1-sided floppy disk. Do not confuse this bit with Bit 4 of DCT+3. This bit shows if the disk is double-sided; Bit 4 of DCT+3 tells the controller what side the current I/O is to be on.

If the hard drive bit (DCT+3, Bit 3) is set, a "1" denotes double the cylinder count stored in DCT+6. (This implies that a logical cylinder is made up of two physical cylinders.)

Bit 4 — If "1," indicates an alien (non-standard) disk controller.

Bits 0-3 — Contain the physical drive address by bit selection (0001, 0010, 0100, and 1000 equal logical Drives 0, 1, 2, and 3, respectively, in a default system). The system supports a translation only where no more than one bit can be set.

If the alien bit (Bit 4) is set, these bits may indicate the starting head number.

DCT+5

Contains the current cylinder position of the drive. It normally stores a copy of the Floppy Disk Controller's track register contents whenever the FDC is selected for access to this drive. It can then be used to reload the track register whenever the FDC is reselected.

If the alien bit (DCT+4, Bit 4) is set, DCT+5 may contain the drive select code for the alien controller.

DCT+6

Contains the highest numbered cylinder on the drive. Since cylinders are numbered from zero, a 35-track drive is recorded as X'22,' a 40-track drive as X'27,' and an 80-track drive as X'4F.' If the hard drive bit (DCT+3, Bit 3) is set, the true cylinder count depends on DCT+4, Bit 5. If that bit is a "1," DCT+6 contains only half of the true cylinder count.

DCT+7

Contains allocation information.

Bits 5-7 — Contain the number of heads for a hard drive.

Bits 0-4 — Contain the highest numbered sector relative to zero. A 10-sector-per-track drive would show X'09.' If DCT+4, Bit 5 indicates 2-sided operation, the sectors per cylinder equals twice this number.

DCT+8

Contains additional allocation information.

Bits 5-7 — Contain the number of granules per track allocated in the formatting process. If DCT+4, Bit 5 indicates 2-sided operation, the granules per cylinder equals twice this number. For a hard drive, this number is the total granules per cylinder.

Bits 0-4 — Contain the number of sectors per granule that was used in the formatting operation.

DCT+9

Contains the number of the cylinder where the directory is located. For any directory access, the system first attempts to use this value to read the directory. If this operation is unsuccessful, the system examines the BOOT granule (cylinder 0) directory address byte.

Bytes DCT + 6, DCT + 7, and DCT + 8 must relate without conflicts. That is, the highest numbered sector (+ 1) divided by the number of sectors per granule (+ 1) must equal the number of granules per track (+ 1).

Disk I/O Table

TRSDOS interfaces with hardware peripherals by means of software drivers. The drivers are, in general, coupled to the operating system through data parameters stored in the system's many tables. In this way, hardware not currently supported by TRSDOS can easily be supported by generating driver software and updating the system tables.

Disk drive sub-systems (such as controllers for 5¼" drives, 8" drives, and hard disk drives) have many parameters addressed in the Drive Code Table (DCT). Besides those operating parameters, controllers also require various commands (SELECT, SECTOR READ, SECTOR WRITE, and so on) to control the physical devices. TRSDOS has defined command conventions to deal with most commands available on standard Disk Controllers.

The function value (hexadecimal or decimal) you wish to pass to the driver should go in register B. The available functions are:

Hex	Dec	Function	Operation Performed
X'00'	0	DCSTAT	Test to see if drive is assigned in DCT
X'01'	1	SELECT	Select a new drive and return status
X'02'	2	DCINIT	Set to cylinder 0, restore, set side 0
X'03'	3	DCRES	Reset the Floppy Disk Controller
X'04'	4	RSTOR	Issue FDC RESTORE command
X'05'	5	STEPI	Issue FDC STEP IN command
X'06'	6	SEEK	Seek a cylinder
X'07'	7	TSTBSY	Test to see if requested drive is busy
X'08'	8	RDHDR	Read sector header information
X'09'	9	RDSEC	Read sector
X'0A'	10	VRSEC	Verify if the sector is readable
X'0B'	11	RDTRK	Issue an FDC track read command
X'0C'	12	HDFMT	Format the device
X'0D'	13	WRSEC	Write a sector
X'0E'	14	WRSYS	Write a system sector (for example, directory)
X'0F'	15	WRTRK	Issue an FDC track write command

Function codes X'10' to X'FF' are reserved for future use.

Directory Records (DIREC)

The directory contains information needed to access all files on the disk. The directory records section is limited to a maximum of 32 sectors because of physical limitations in the Hash Index Table. Two additional sectors in the directory cylinder are used by the system for the Granule Allocation Table and the Hash Index Table. The directory is contained on one cylinder. Thus, a 10-sector-per-cylinder formatted disk has, at most, eight directory sectors. See the section on the Hash Index Table for the formula to calculate the number of directory sectors.

A directory record is 32 bytes in length. Each directory sector contains eight directory records (256/32=8). On system disks, the first two directory records of the first eight directory sectors are reserved for system overlays. The total number of files possible on a disk equals the number of directory sectors times eight (since 256/32=8). The number available for use is reduced by 16 on system disks to account for those record slots reserved for the operating system. The following table shows the directory record capacity (file capacity) of each format type. The dash suffix (-1 or -2) on the items in the density column represents the number of sides formatted (for example, SDEN-1 means single density, 1-sided).

	Sectors per Cylinder	Directory Sectors	User Files on Data Disk**	User Files on SYS Disk
5" SDEN-1	10	8	62	48
5" SDEN-2	20	18	142	128
5" DDEN-1	18	16	126	112
5" DDEN-2	36	32	254	240
8" SDEN-1	16	14	110	96
8" SDEN-2	32	30	238	224
8" DDEN-1	30	28	222	208
8" DDEN-2	60	32	254	240
Hard Disk*				

*Hard drive format depends on the drive size and type, as well as the user's division of the physical drive into logical drives. After setting up and formatting the drive, you can use the FREE library command to see the available files.

****Note:** Two directory records are reserved for BOOT/SYS and DIR/SYS, and are included in the figures for this column.

TRSDOS Version 6 is upward compatible with other TRSDOS 2.3 compatible operating systems in its directory format. The data contained in the directory has been extended. An SVC is included to either display an abbreviated directory or place its data in a user-defined buffer area. For detailed information, see the @DODIR and @RAMDIR SVCs.

The following information describes the contents of each directory field:

DIR+0

Contains all attributes of the designated file.

Bit 7 — If "0," this flag indicates that the directory record is the file's primary directory entry (FPDE). If "1," the directory record is one of the file's extended directory entries (FXDE). Since a directory entry can contain information on up to four extents (see notes on the extent fields, beginning with DIR+22), a file that is fractured into more than four extents requires additional directory records.

Bit 6 — Specifies a SYStem file if "1," a nonsystem file if "0."

Bit 5 — If set to "1," indicates a Partition Data Set (PDS) file.

Bit 4 — Indicates whether the directory record is in use or not. If set to "1," the record is in use. If "0," the directory record is not active, although it may appear to contain directory information. In contrast to some operating systems that zero out the directory record when you remove a file, TRSDOS only resets this bit to zero.

Bit 3 — Specifies the visibility. If "1," the file is INVisible to a directory display or other library function where visibility is a parameter. If a "0," then the file is VISible. (The file can be referenced if specified by name by an @INIT or @OPEN SVC.)

Bits Ø-2—Contain the USER protection level of the file. The 3-bit binary value is one of the following:

Ø = FULL	2 = RENAME	4 = UPDATE	6 = EXECUTE
1 = REMOVE	3 = WRITE	5 = READ	7 = NO ACCESS

DIR + 1

Contains various file flags and the month field of the packed date of last modification.

Bit 7—Set to "1" if the file was "CREATEd" (see CREATE library command in the *Disk System Owner's Manual*). Since the CREATE command can reference a file that is currently existing but non-CREATEd, it can turn a non-CREATEd file into a CREATEd one. You can achieve the same effect by changing this bit to a "1."

Bit 6—If set to "1," the file has not been backed up since its last modification. The BACKUP utility is the only TRSDOS facility that resets this flag. It is set during the close operation if the File Control Block (FCB + Ø, Bit 2) shows a modification of file data.

Bit 5—If set to "1," indicates a file in an open condition with UPDATE access or greater.

Bit 4—If the file was modified during a session where the system date was not maintained, this bit is set to "1." This specifies that the packed date of modification (if any) stored in the next three fields is not the actual date the modification occurred. If this bit is "1," the directory command displays plus signs (+) between the date fields.

Bits Ø-3—Contain the binary month of the last modification date. If this field is a zero, DATE was not set when the file was established or since if it was updated.

DIR + 2

Contains the remaining date of modification fields.

Bits 3-7—Contain the binary day of last modification.

Bits Ø-2—Contain the binary year minus 8Ø. For example, 198Ø is coded as ØØØ, 1981 as ØØ1, 1982 as Ø1Ø, and so on.

DIR + 3

Contains the end-of-file offset byte. This byte and the ending record number (ERN) form a pointer to the byte position that follows the last byte written. This assumes that programmers, interfacing in machine language, properly maintain the next record number (NRN) offset pointer when the file is closed.

DIR + 4

Contains the logical record length (LRL) specified when the file was generated or when it was later changed with a CLONE parameter.

DIR + 5 through DIR + 12

Contain the name field of the filespec. The filename is left justified and padded with trailing blanks.

DIR + 13 through DIR + 15

Contain the extension field of the filespec. It is left justified and padded with trailing blanks.

DIR + 16 and DIR + 17

Contain the OWNER password hash code.

DIR + 18 and DIR + 19

Contain the USER password hash code. The protection level in DIR + Ø is associated with this password.

DIR+20 and DIR+21

Contain the ending record number (ERN), which is based on full sectors. If the ERN is zero, it indicates that no writing has taken place (or that the file was not closed properly). If the LRL is not 256, the ERN represents the sector where the EOF occurs. You should use ERN minus 1 to account for a value relative to sector 0 of the file.

DIR+22 and DIR+23

This is the first extent field. Its contents indicate which cylinder stores the first granule of the extent, which relative granule it is, and how many contiguous grans are in use in the extent.

DIR+22 — Contains the cylinder value for the starting gran of that extent.

DIR+23, Bits 5-7 — Contain the number of the granule in the cylinder indicated by DIR+22 which is the first granule of the file for that extent. This value is relative to zero ("0" denotes the first gran, "1" denotes the second, and so on).

DIR+23, Bits 0-4 — Contain the number of contiguous granules, relative to 0 ("0" denotes one gran, "1" denotes two, and so on). Since the field is five bits, it contains a maximum of X'1F' or 31, which represents 32 contiguous grans.

DIR+24 and DIR+25

Contain the fields for the second extent. The format is identical to that for Extent 1.

DIR+26 and DIR+27

Contain the fields for the third extent. The format is identical to that for Extent 1.

DIR+28 and DIR+29

Contain the fields for the fourth extent. The format is identical to that for Extent 1.

DIR+30

This is a flag noting whether or not a link exists to an extended directory record. If no further directory records are linked, the byte contains X'FF.' A value of X'FE' in this byte establishes a link to an extended directory entry. (See "Extended Directory Records" below.)

DIR+31

This is the link to the extended directory entry noted by the previous byte. The link code is the Directory Entry Code (DEC) of the extended directory record. The DEC is actually the position of the Hash Index Table byte mapped to the directory record. For more information, see the section "Hash Index Table."

Extended Directory Records

Extended directory records (FXDE) have the same format as primary directory records, except that only Bytes 0, 1, and 21-31 are utilized. Within Byte 0, only Bits 4 and 7 are significant. Byte 1 contains the DEC of the directory record of which this is an extension. An extended directory record may point to yet another directory record, so a file may contain an "unlimited" number of extents (limited only by the total number of directory records available).

Granule Allocation Table (GAT)

The Granule Allocation Table (GAT) contains information on the free and assigned space on the disk. The GAT also contains data about the formatting used on the disk.

A disk is divided into cylinders (tracks) and sectors. Each cylinder has a specified number of sectors. A group of sectors is allocated whenever additional space is needed. This group is called a granule. The number of sectors per granule depends on the total number of sectors available on a logical drive. The GAT provides for a maximum of eight granules per cylinder.

In the GAT bytes, each bit set to "1" indicates a corresponding granule in use (or locked out). Each bit reset to "0" indicates a granule free to be used. In a GAT byte, bit 0 corresponds to the first relative granule, bit 1 to the second relative granule, bit 2 the third, and so on. A 5¼" single density diskette is formatted at 10 sectors per cylinder, 5 sectors per granule, 2 granules per cylinder. Thus, that configuration uses only bits 0 and 1 of the GAT byte. The remainder of the GAT byte contains all 1's, denoting unavailable granules. Other formatting conventions are as follows:

	Sectors per Cylinder	Sectors per Granule	Granules per Cylinder	Maximum No. of Cylinders
5" SDEN	10	5	2	80
5" DDEN	18	6	3	80
8" SDEN	16	8	2	77
8" DDEN	30	10	3	77
Hard Disk	32	16	8	153

*Hard drive format depends on the drive size and type, as well as the user's division of the drive into logical drives. These values assume that one physical hard disk is treated as one logical drive.

The above table is valid for single-sided disks. TRSDOS supports double-sided operation if the hardware interfacing the physical drives to the CPU allows it. A two-headed drive functions as a single logical drive, with the second side as a cylinder-for-cylinder extension of the first side. A bit in the Drive Code Table (DCT + 4, Bit 5) indicates one-sided or two-sided drive configuration.

A Winchester-type hard disk can be divided by heads into multiple logical drives. Details are supplied with Radio Shack drives.

The Granule Allocation Table is the first relative sector of the directory cylinder. The following information describes the layout and contents of the GAT.

GAT + X'00' through GAT + X'5F'

Contains the free/assigned table information. GAT + 0 corresponds to cylinder 0, GAT + 1 corresponds to cylinder 1, GAT + 2 corresponds to cylinder 2, and so on. As noted above, bit 0 of each byte corresponds to the first granule on the cylinder, bit 1 to the second granule, and so on. A value of "1" indicates the granule is not available for use.

GAT + X'60' through GAT + X'BF'

Contains the available/locked out table information. It corresponds cylinder for cylinder in the same way as the free/assigned table. It is used during mirror-image backups to determine if the destination diskette has the proper capacity to effect a backup of the source diskette. This table does not exist for hard disks; for this reason, mirror-image backups cannot be performed on hard disk.

GAT + X'C0' through GAT + X'CA'

Used in hard drive configurations; extends the free/assigned table from X'00' through X'CA'. Hard drive capacity up to 203 (0-202) logical or 406 physical cylinders is supported.

GAT + X'CB'

Contains the operating system version that was used in formatting the disk. For example, disks formatted under TRSDOS 6.2 have a value of X'62' contained in this byte. It is used to determine whether or not the disk contains all of the parameters needed for TRSDOS operation.

GAT+X'CC'

Contains the number of cylinders in excess of 35. It is used to minimize the time required to compute the highest numbered cylinder formatted on the disk. It is excess 35 to provide compatibility with alien systems not maintaining this byte. If you have a disk that was formatted on an alien system for other than 35 cylinders, this byte can be automatically configured by using the REPAIR utility. (See the section on the REPAIR utility in the *Disk System Owner's Manual.*)

GAT+X'CD'

Contains data about the formatting of the disk.

Bit 7 — If set to "1," the disk is a data disk. If "0," the disk is a system disk.

Bit 6 — If set to "1," indicates double-density formatting. If "0," indicates single-density formatting.

Bit 5 — If set to "1," indicates 2-sided disk. If "0," indicates 1-sided disk.

Bits 3-4 — Reserved.

Bits 0-2 — Contain the number of granules per cylinder minus 1.

GAT+X'CE' and GAT+X'CF'

Contain the 16-bit hash code of the disk master password. The code is stored in standard low-order, high-order format.

GAT+X'D0' through GAT+X'D7'

Contain the disk name. This is the name displayed during a FREE or DIR operation. The disk name is assigned during formatting or during an ATTRIB disk renaming operation. The name is left justified and padded with blanks.

GAT+X'D8' through GAT+X'DF'

Contain the date that the diskette was formatted or the date that it was used as the destination in a mirror image backup operation in the format mm/dd/yy.

GAT+X'E0' through GAT+X'FF'

Reserved for system use.

In Version 6.2:

GAT+X'E0' through GAT+X'F4'

Reserved for system use.

GAT+X'F5' through GAT+X'FF'

Contain the Media Data Block (MDB).

GAT+X'F5' through GAT+X'F8' — the identifying header. These four bytes contain a 3 (X'03'), followed by the letters LSI (X'4C',X'53',X'49').

GAT + X'F8' through GAT9 + X'FF' — the last seven bytes of the DCT in use when the media was formatted. FORMAT, MemDISK, and TRSFORM6 install this information. See Drive Control Table (DCT) for more information on these bytes.

Hash Index Table (HIT)

The Hash Index Table is the key to addressing any file in the directory. It pinpoints the location of a file's directory with a minimum of disk accesses, keeping overhead low and providing rapid file access.

The system's procedure is to construct an 11-byte filename/extension field. The filename is left-justified and padded with blanks. The file extension is then inserted and padded with blanks; it occupies the three least significant bytes of the 11-byte field. This field is processed through a hashing algorithm which produces a single byte value in the range X'01' through X'FF'. (A hash value of X'00' indicates a spare HIT position.)

The system then stores the hash code in the Hash Index Table (HIT) at a position corresponding to the directory record that contains the file's directory. Since more than one 11-byte string can hash to identical codes, the opportunity for "collisions" exists. For this reason, the search algorithm scans the HIT for a matching code entry, reads the directory record corresponding to the matching HIT position, and compares the filename/extension stored in the directory with that provided in the file specification. If both match, the directory has been found. If the two fields do not match, the HIT entry was a collision and the algorithm continues its search from the next HIT entry.

The position of the HIT entry in the hash table is called the Directory Entry Code (DEC) of the file. All files have at least one DEC. Files that are extended beyond four extents have a DEC for each extended directory entry and use more than one filename slot. To maximize the number of file slots available, you should keep your files below five extents where possible.

Each HIT entry is mapped to the directory sectors by the DEC's position in the HIT. Think of the HIT as eight rows of 32-byte fields. Each row is mapped to one of the directory records in a directory sector: The first HIT row is mapped to the first directory record, the second HIT row to the second directory record, and so on. Each column of the HIT field (0-31) is mapped to a directory sector. The first column is mapped to the first directory sector in the directory cylinder (not including the GAT and HIT). Therefore, the first column corresponds to sector 2, the second column to sector 3, and so on. The maximum number of HIT columns used depends on the disk formatting according to the formula: N = number of sectors per cylinder minus two, up to 32.

The following chart shows the correlation of the Hash Index Table to the directory records. Each byte value shown represents the position in the HIT. This position value is the DEC. The actual contents of each byte is either a X(00) indicating a spare slot, or the 1-byte hash code of the file that occupies the corresponding directory record.

	Columns															
Row 1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
Row 2	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
Row 3	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
Row 4	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
Row 5	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
Row 6	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
Row 7	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
Row 8	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

A 5¼" single density disk has 10 sectors per cylinder, two of which are reserved for the GAT and HIT. Since only eight directory sectors are possible, only the first eight positions of each HIT row are used. Other formats use more columns of the HIT, depending on the number of sectors per cylinder in the formatting scheme.

The eight directory records for sector 2 of the directory cylinder correspond to assignments in HIT positions 00, 20, 40, 60, 80, A0, C0, and E0. On system disks, the following positions are reserved for system overlays. On data disks, these positions (except for 00 and 01) are available to the user.

00 — BOOT/SYS	20 — SYS6/SYS
01 — DIR/SYS	21 — SYS7/SYS
02 — SYS0/SYS	22 — SYS8/SYS
03 — SYS1/SYS	23 — SYS9/SYS
04 — SYS2/SYS	24 — SYS10/SYS
05 — SYS3/SYS	25 — SYS11/SYS
06 — SYS4/SYS	26 — SYS12/SYS
07 — SYS5/SYS	27 — SYS13/SYS

These entry positions correspond to the first two rows of each directory sector for the first eight directory sectors. Since the operating system accesses these overlays by position in the HIT rather than by filename, these positions are reserved on system disks.

The design of the Hash Index Table limits the number of files on any one drive to a maximum of 256.

Locating a Directory Record

Because of the coding scheme used on the entries in the HIT table, you can locate a directory record with only a few instructions. The instructions are:

```
AND   1FH
ADD   A,2
```

(calculates the sector)

and

```
AND   0E0H
```

(calculates the offset in that sector)

For example, if you have a Directory Entry Code (DEC) of X'84', the following occurs when these instructions are performed:

	Value of accumulator
	A=X'84'
AND 1FH	
	A=X'04'
ADD A,2	
	A=X'06' The record is in the seventh sector of the directory cylinder (0-6)

Using the Directory Entry Code (DEC) again, you can find the offset into the sector that was found using the above instructions by executing one instruction:

	Value of accumulator
	A=X'84'
AND 0E0H	
	A=X'80' The directory record is X'80' (128) bytes from the beginning of the sector

If the record containing the sector is loaded on a 256-byte boundary (LSB of the address is X'00') and HL points to the starting address of the sector, then you can use the above value to calculate the actual address of the directory record by executing the instruction:

```
LD    L,A
```

When executed after the calculation of the offset, this causes HL to point to the record. For example:

```
A=X'80'
            LD     HL,4200H   ;Where sector is loaded
            LD     L,A        ;Replace LSB with offset
```

HL now contains 4280H, which is the address of the directory record you wanted.

If you cannot place the sector on a 256-byte boundary, then you can use the following instructions:

```
A=X'80'
            LD     HL,4256H   ;Where sector is loaded
            LD     E,A        ;Put offset in E (LSB)

            LD     D,0        ;Put a zero in D (MSB)
            ADD    HL,DE      ;Add two values together
```

HL now contains 42D6H, which is the address of the directory record.

Note that the first DEC found with a matching hash code may be the file's extended directory entry (FXDE). Therefore, if you are going to write system code to deal with this directory scheme, you must properly deal with the FPDE/FXDE entries. See Directory Records for more information.

File Control Block (FCB)

The File Control Block (FCB) is a 32-byte memory area. Before the file is opened, this space holds the file's filespec. After an @OPEN or @INIT supervisor call is performed, the system uses this area to interface with the file, and replaces the filespec with other information. When the file is closed, the filespec (without any specified password) is returned to the FCB.

While a file is open, the contents of the FCB are dynamic. As records are written to or read from the disk file, specific fields in the FCB are modified. Avoid changing the contents of the FCB during the time a file is open, unless you are sure that the change will not affect the integrity of the file.

During most system access of the FCB, the IX index register is used to reference each field of data. Register pair DE is used mainly for the initial reference to the FCB address. The information contained in each field of the FCB is as follows:

FCB+0

Contains the TYPE code of the control block.

Bit 7 — If set to "1," indicates that the file is in an open condition; if "0," the file is assumed closed. This bit can be tested to determine the "open" or "closed" status of an FCB.

Bit 6 — Is set to "1" if the file was opened with UPDATE access or higher.

Bit 5 — Indicates a Partition Data Set (PDS) type file.

Bits 4-3 — Reserved for future use.

Bit 2 — Is set to "1" if the system performed any WRITE operation on this file. It is used to update the MOD flag in the directory record when the file is closed.

Bits 1-0 — Reserved for future use.

FCB+1

Contains status flag bits used in read/write operations by the system.

Bit 7 — If set to "1," indicates that I/O operations will be either full sector operations or byte operations of logical record length (LRL) less than 256. If "0," only sector operations will be performed. If you are going to use only full-sector I/O, you can reduce system overhead by specifying the LRL at open time as 0 (indicating 256). An LRL of other than 256 sets bit 7 to "1" on open.

Bit 6 — If set to "1," indicates that the end of file (EOF) is to be set to ending record number (ERN) only if next record number (NRN) exceeds the current value of EOF. This is the case if random access is to be used. During random access, the EOF is not disturbed unless you extend the file beyond the last record slot. Any time the position routine (@POSN) is called, bit 6 is automatically set. If bit 6 is "0," then EOF will be updated on every WRITE operation.

Bit 5 — If "0," then the disk I/O buffer contains the current sector denoted by NRN. If set to "1," then the buffer does not contain the current sector. During byte I/O, bit 5 is set when the last byte of the sector is read. A sector read resets the bit, showing the buffer to be current.

Bit 4 — If set to "1," indicates that the buffer contents have been changed since the buffer was read from the file. It is used by the system to determine whether the buffer must be written back to the file before reading another record. If "0," then the buffer contents were not changed.

Bit 3 — Used to specify that the directory record is to be updated each time the NRN exceeds the EOF. (The normal operation is to update the directory only when an FCB is closed.) Some unattended operations may use this extra measure of file protection. It is specified by adding an exclamation mark ("!") to the end of a filespec when the filespec is requested at open time.

Bits 2-0 — Contain the user (access) protection level as retrieved from the directory of the file. The 3-bit binary value is one of the following:

0 = FULL	2 = RENAME	4 = UPDATE	6 = EXECUTE
1 = REMOVE	3 = WRITE	5 = READ	7 = NO ACCESS

FCB + 2

Used by Partition Data Set (PDS) files.

FCB + 3 and FCB + 4

Contain the buffer address in low-order, high-order format. This is the buffer address specified in register pair HL when the @INIT or @OPEN SVC is performed.

FCB + 5

Contains the relative byte offset within the current buffer for the next I/O operation. If this byte has a zero value, then FCB + 1, Bit 5 must be examined to see if the first byte in the current buffer is the target position or if it is the first byte of the next record. If you are performing sector I/O of byte data (that is, maintaining your own buffering), then it is important to maintain this byte when you close the file if the true end of file is not at a sector boundary.

FCB + 6

Bits 3-7 — Reserved for system use.

Bits 0-2 — Contain the logical drive number in binary of the drive containing the file. Do not modify this byte; altering this value may damage other files. This byte and FCB + 7 are the only links to the file's directory information.

FCB + 7

Contains the directory entry code (DEC) for the file. This code is the offset in the Hash Index Table where the hash code for the file appears. Do not modify this byte; altering this value may damage other files. This byte and FCB + 6 are the only links to the directory information for the file.

FCB + 8

Contains the end-of-file byte offset. This byte is similar to FCB + 5 except that it pertains to the end of file rather than to the next record number.

FCB + 9

Contains the logical record length that was in effect when the file was opened. This may not be the same LRL that exists in the directory. The directory LRL is generated at the file creation and never changes unless the file is overwritten.

FCB + 10 and FCB + 11

Contain the next record number (NRN), which is a pointer for the next I/O operation. When a file is opened, NRN is zero, indicating a pointer to the beginning. Each sequential sector I/O advances NRN by one.

FCB + 12 and FCB + 13

Contain the ending record number (ERN) of the file. This is a pointer to the sector that contains the end-of-file indicator. In a null file (one with no records), ERN equals 0. If one sector has been written, ERN equals 1.

FCB + 14 and FCB + 15

Contain the same information as the first extent of the directory. This represents the starting cylinder of the file (FCB + 14) and the starting relative granule within the starting cylinder (FCB + 15). FCB + 15 also contains the number of contiguous granules allocated in the extent. These bytes are used as a pointer to the beginning of the file referenced by the FCB.

FCB + 16 through FCB + 19

This 4-byte entry contains granule allocation information for an extent of the file. Relative bytes 0 and 1 contain the total number of granules allocated to the file up to but not including the extent referenced by this field. Relative byte 2 contains the starting cylinder of this extent. Relative byte 3 contains the starting relative granule for the extent and the number of contiguous granules.

FCB + 20 through FCB + 23

Contain information similar to the above but for a second extent of the file.

FCB + 24 through FCB + 27

Contain information similar to the above but for a third extent of the file.

FCB + 28 through FCB + 31

Contain information similar to the above but for a fourth extent of the file.

The file control block contains information on only four extents at one time. If the file has more than four extents, additional directory accessing is done to shift the 4-byte entries in order to make space for the new extent information.

Although the system can handle a file of any number of extents, you should keep the number of extents small. The most efficient file is one with a single extent. The number of extents can be reduced by copying the file to a disk that contains a large amount of free space.

Converting to TRSDOS Version 6

This section provides suggestions on writing programs effectively with TRSDOS Version 6, and on converting programs created with TRSDOS 1.3 and LDOS 5.1 operating systems for use with TRSDOS Version 6. This information is by no means complete, but presents some important concepts to keep in mind when using TRSDOS Version 6.

When programming in assembly language, you can use TRSDOS Version 6 routines for commonly used operations. These are accessed through the supervisor calls (SVCs) instead of absolute call addresses. Nothing in the system can be accessed via any absolute address reference (except Z-80 RST and NMI jump vectors).

IMPORTANT NOTE: TRSDOS provides all functions and storage through supervisor calls. No address or entry point below 3000H is documented or supported by Radio Shack.

The keyboard is not accessible via "peeking," and the video RAM cannot be "poked." The keyboard and video are accessible only through the appropriate SVCs.

Another distinction is that TRSDOS Version 6 handling of logical byte I/O devices (keyboard, video, printer, communications line) completely supports error status feedback. A FLAG convention is uniform throughout these device drivers as well as physical byte I/O associated with files. The device handling in TRSDOS Version 6 is completely independent. That means that byte I/O, both logical and physical, can be routed, filtered, and linked. Therefore, it is important to test status return codes in all applications using byte I/O regardless of the device that the application expects to be used, since re-direction to some other device is possible at the TRSDOS level. Appropriate action must be taken when errors are detected.

Modules loaded into memory and protected by lowering HIGH$ must include the standard header, as described earlier under "Memory Header." The @GTMOD supervisor call requires that this header be present in every resident module for proper operation.

The file password protection terms of UPDATE and ACCESS have been changed in TRSDOS Version 6 to OWNER and USER, respectively. The additional file protection level of UPDATE has been added. A file with UPDATE protection level can be read or written to, but its end of file cannot be extended. This protection can be useful in a random access fixed-size file or in a file where shared access is to take place.

Files opened with UPDATE or greater access are indicated as open in their directory. Attempting to open the file again forces a change to READ access protection and a "File already open" error code. It is therefore important for applications to CLOSE files that are opened.

For the convenience of applications that access files only for reading, you can inhibit the "file open bit." If you set bit 0 of the system flag SFLAG$ (see the @FLAGS supervisor call), the file open bit is not set in the file's directory. Once set, the next @OPEN or @INIT SVC automatically resets bit 0 of SFLAG$. Note that you cannot use this procedure for files being written to, since it inhibits the CLOSE process.

Some application programs need access to certain system parameters and variables. A number of flags, variables, and port images can be accessed relative to a flag pointer obtained via the @FLAGS supervisor call. These parameters are only accessible relative to this pointer, as the pointer's location may change. (See the explanation of the @FLAGS SVC.)

All applications must honor the contents of HIGH$. This pointer contains the highest RAM address usable by any program. You can retrieve and change HIGH$ by using the @HIGH$ SVC.

TRSDOS Version 6 library commands and utilities supply a return code (RC) at completion. The RC is returned in register pair HL. The value returned is either zero (indicating no error), a number from one through 62 (indicating an error as noted in Appendix A, TRSDOS Error Messages), or X'FFFF' (indicating an extended error which is currently not assigned an error number). TRSDOS Version 6 Job Control Language (JCL) aborts on any program terminating with a non-zero RC value. Applications should therefore properly set the return code register pair HL before exiting.

TRSDOS Version 6 library commands are also invokable via the @CMNDR SVC which executes the command. Library commands properly maintain the Stack Pointer (SP) and exit via a RET instruction. In this manner, control is returned to the invoking program with the RC present for testing. For commands invoked with the @CMNDI SVC or prompted for via the @EXIT SVC, the SP is restored to the system stack. The top of the stack will contain an address suitable for simulating an @EXIT SVC; thus, if your application program properly maintains the integrity of the stack pointer, it can exit after setting the RC via a RET instruction instead of an @EXIT SVC.

TRSDOS Version 6 diskette and file structure is identical to that used in LDOS 5.1. This includes formatting, directory structure, and data address mark conventions. TRSDOS Version 6 system diskettes, however, use the entire BOOT track (track 0). This compatibility means that data files may be used interchangeably between LDOS 5.1 equipped machines and TRSDOS Version 6 equipped machines; the diskettes themselves are readable and writable across both operating systems.

The methods of internal handling of device linking and filtering have been changed from LDOS 5.1. (It is beyond the scope of this manual to explain the internal functioning of TRSDOS Version 6.) Device filters must adhere to a strict protocol of linkage in order to function properly. See the section on "Device Driver and Filter Templates" for information on device driver and filter protocol.

Stack Handling Restrictions*

Interrupt tasks and filters that deal with the keyboard or video must not place the stack pointer above X'F3FF'. This is because any operation that requires the keyboard or video RAM switches in the 3K bank at X'F400' and suppresses the stack until it is switched out again. If the system accesses the stack at any time during this period, the integrity of the stack is destroyed.

*In TRSDOS 6.0.0, the stack cannot be placed above X'F3FF' for any reason.

Programming With Restart Vectors

The Restart instruction (RST) provides the assembly language programmer with the ability to call a subroutine with a one-byte call. If a routine is called many times by a program, the amount of space that is saved by using the RST instruction (instead of a three-byte CALL) can be significant.

In TRSDOS a RST instruction is also used to interface to the operating system. The system uses RST 28H for supervisor calls. RSTS 00H, 30H, and 38H are for the system's internal use.

RSTs 08H, 10H, 18H, and 20H are available for your use. Caution: Some programs, such as BASIC, may use some of these RSTs.

Each RST instruction calls the address given in the operand field of the instruction. For example, RST 18H causes the system to push the current program counter address onto the stack and then set the program counter to address 0018H. RST 20H causes a jump to location 0020H, and so on.

Each RST has three bytes reserved for the subroutine to use. If the subroutine will not fit in three bytes, then you should code a jump instruction (JP) to where the subroutine is located. At the end of the subroutine, code a return instruction (RET). Control is then transferred to the instruction that follows the RST.

For example, suppose you want to use RST 18H to call a subroutine named "ROUTINE." The following routine loads the restart vector with a jump instruction and saves the old contents of the restart vector for later use.

```
SETRST:  LD      IX,0018H    ;Restart area address
         LD      IY,RDATA    ;Data area address
         LD      B,3         ;Number of bytes to move
LOOP:    LD      A,(IX)      ;Read a byte from
                             ;restart area
         LD      C,(IY)      ;Read a byte from data
                             ;area
         LD      (IX),C      ;Store this byte in
                             ;restart area
         LD      (IY),A      ;Store this byte in data
                             ;area
         INC     IX          ;Increment restart area
                             ;pointer
         INC     IY          ;Increment data area
                             ;pointer
         DJNZ    LOOP        ;Loop till 3 bytes moved
         RET                 ;Return when done
RDATA:   DEFB    0C3H        ;Jump instruction (JP)
         DEFW    ROUTINE     ;Operand (name of
                             ;subroutine)
```

Before exiting the program, calling the above routine again puts the original contents of the restart vector back in place.

KFLAG$ (BREAK), (PAUSE), and (ENTER) Interfacing

KFLAG$ contains three bits associated with the keyboard functions of BREAK, PAUSE ((SHIFT) (@)), and ENTER. A task processor interrupt routine (called the KFLAG$ scanner) examines the physical keyboard and sets the appropriate KFLAG$ bit if any of the conditions are observed. Similarly, the RS-232C driver routine also sets the KFLAG$ bits if it detects the matching conditions being received.

Many applications need to detect a PAUSE or BREAK while they are running. BASIC checks for these conditions after each logical statement is executed (that is, at the end of a line or at a ":"). That is how, in BASIC, you can stop a program with the (BREAK) key or pause a listing.

One method of detecting the condition in previous TRSDOS operating systems was to issue the @KBD supervisor call to check for BREAK or PAUSE ((SHIFT)(@)), ignoring all other keys. Unfortunately, this caused keyboard type-ahead to be ineffective; the @KBD SVC flushed out the type-ahead buffer if any other keystrokes were stacked up.

Another method was to scan the keyboard, physically examining the keyboard matrix. An undesirable side effect of this method was that type-ahead stored up the keyboard depression for some future unexpected input request. Examining the keyboard directly also inhibits remote terminals from passing the BREAK or PAUSE condition.

In TRSDOS Version 6, the KFLAG$ scanner examines the keyboard for the BREAK, PAUSE, and ENTER functions. If any of these conditions are detected, appropriate bits in the KFLAG$ are set (bits 0, 1, and 2 respectively).

Note that the KFLAG$ scanner only sets the bits. It does not reset them because the "events" would occur too fast for your program to detect. Think of the KFLAG$ bits as a latch. Once a condition is detected (latched), it remains latched until something examines the latch and resets it—a function to be performed by your KFLAG$ detection routine.

Under Version 6.2, you can use the @CKBRKC SVC, SVC 106, to see if the BREAK key has been pressed. If a BREAK condition exists, @CKBRKC resets the break bit of KFLAG$.

For illustration, the following example routine uses the BREAK and PAUSE conditions:

```
KFLAG$   EQU   10
@FLAGS   EQU   101
@KBD     EQU   8
@KEY     EQU   1
@PAUSE   EQU   16
CKPAWS   LD    A,@FLAGS         ;Get Flags pointer
         RST   28H              ;into register IY
         LD    A,(IY+KFLAG$)    ;Get the KFLAG$
         RRCA                   ;Bit 0 to carry
         JP    C,GOTBRK         ;Go on BREAK
         RRCA                   ;Bit 1 to carry
         RET   NC               ;Return if no pause
         CALL  RESKFL           ;Reset the flag
         PUSH  DE
FLUSH    LD    A,@KBD           ;Flush type-ahead
         RST   28H              ;buffer while
         JR    Z,FLUSH          ;ignoring errors
         POP   DE
PROMPT   PUSH  DE
         LD    A,@KEY           ;Wait on key entry
         RST   28H
         POP   DE
         CP    80H              ;Abort on (BREAK)
         JP    Z,GOTBRK
         CP    60H              ;Ignore PAUSE;
         JR    Z,PROMPT         ;else . . .
RESKFL   PUSH  HL               ;reset KFLAG$
         PUSH  AF
         LD    A,@FLAGS         ;Get flags pointer
         RST   28H              ;into register IY
RESKFL1  LD    A,(IY+KFLAG$)    ;Get the flag
         AND   0F8H             ;Strip ENTER,
```

```
        LD      (IY+KFLAG$),A ;PAUSE, BREAK
        PUSH    BC
        LD      B,16
        LD      A,@PAUSE      ;Pause a while
        RST     28H
        POP     BC
        LD      A,(IY+KFLAG$) ;Check if finger is
        AND     3             ;still on key
        JR      NZ,RESKFL1    ;Reset it again
        POP     AF            ;Restore registers
        POP     HL            ;and exit
        RET
```

The best way to explain this KFLAG$ detection routine is to take it apart and discuss each subroutine. The first piece reads the KFLAG$ contents:

```
KFLAG$  EQU     10
CKPAWS  LD      A,@FLAGS      ;Get Flags pointer
        RST     28H           ;into register IY
        LD      A,(IY+KFLAG$) ;Get the KFLAG$
        RRCA                  ;Bit 0 to carry
        JP      C,GOTBRK      ;Go on BREAK
        RRCA                  ;Bit 1 to carry
        RET     NC            ;Return if no pause
```

The @FLAGS SVC obtains the flags pointer from TRSDOS. Note that if your application uses the IY index register, you should save and restore it within the CKPAWS routine. (Alternatively, you could use @FLAGS to calculate the location of KFLAG$, use register HL instead of IY, and place the address into the LD instructions of CKPAWS at the beginning of your application.)

The first rotate instruction places the BREAK bit into the carry flag. Thus, if a BREAK condition is in effect, the subroutine branches to "GOTBRK," which is your BREAK handling routine.

If there is no BREAK condition, the second rotate places what was originally in the PAUSE bit into the carry flag. If no PAUSE condition is in effect, the routine returns to the caller.

This sequence of code gives a higher priority to BREAK (that is, if both BREAK and PAUSE conditions are pending, the BREAK condition has precedence). Note that the GOTBRK routine needs to clear the KFLAG$ bits after it services the BREAK condition. This is easily done via a call to RESKFL.

The next part of the routine is executed on a PAUSE condition:

```
        CALL    RESKFL        ;Reset the flag
        PUSH    DE
FLUSH   LD      A,@KBD        ;Flush type-ahead
        RST     28H           ;buffer while
        JR      Z,FLUSH       ;ignoring errors
        POP     DE
```

First the KFLAG$ bits are reset via the call to RESKFL. Next, the routine takes care of the possibility that type-ahead is active. If it is, the PAUSE key was probably detected by the type-ahead routine and so is stacked in the type-ahead buffer also. To flush out (remove all stored characters from) the type-ahead buffer, @KBD is called until no characters remain (an NZ is returned).

Now that a PAUSEd state exists and the type-ahead buffer is cleared, the routine waits for a key input:

```
PROMPT  PUSH    DE
        LD      A,@KEY        ;Wait on key entry
        RST     28H
        POP     DE
        CP      80H           ;Abort on (BREAK)
        JP      Z,GOTBRK
```

```
        CP      60H             ;Ignore PAUSE;
        JR      Z,PROMPT        ;else . . .
```

The PROMPT routine accepts a BREAK and branches to your BREAK handling routine. It ignores repeated PAUSE (the 60H). Any other character causes it to fall through to the following routine which clears the KFLAG$:

```
RESKFL  PUSH    HL              ;reset KFLAG$
        PUSH    AF
        LD      A,@FLAGS        ;Get flags pointer
        RST     28H             ;into register IY
RESKFL1 LD      A,(IY+KFLAG$)   ;Get the flag
        AND     0F8H            ;Strip ENTER,
        LD      (IY+KFLAG$),A   ;PAUSE, BREAK
        PUSH    BC
        LD      B,16
        LD      A,@PAUSE        ;Pause a while
        RST     28H
        POP     BC
        LD      A,(IY+KFLAG$)   ;Check if finger is
        AND     3               ;still on key
        JR      NZ,RESKFL1      ;Reset it again
        POP     AF              ;Restore registers
        POP     HL              ;and exit
        RET
```

The RESKFL subroutine should be called when you first enter your application. This is necessary to clear the flag bits that were probably in a "set" condition. This "primes" the detection. The routine should also be called once a BREAK, PAUSE, or ENTER condition is detected and handled. (You need to deal with the flag bits for only the conditions you are using.)

Interfacing to @ICNFG

With the TRSDOS library command SYSGEN, many users may wish to SYSGEN the RS-232C driver. Before doing that, the RS-232C hardware (UART, Baud Rate Generator, etc.) must be initialized. Simply using the SYSGEN command with the RS-232C driver resident is not enough; some initialization routine is necessary. The @ICNFG (Initialization CoNFiGuration) vector is included in TRSDOS to provide a way to invoke a routine to initialize the RS-232C driver when the system is booted. It also provides a way to initialize the hard disk controller at power-up (required by the Radio Shack hard disk system).

The final stages of the booting process loads the configuration file CONFIG/SYS if it exists. After the configuration file is loaded, an initialization subroutine CALLs the @ICNFG vector. Thus, any initialization routine that is part of a memory configuration can be invoked by chaining into @ICNFG.

If you need to configure your own routine that requires initialization at power-up, you can chain into @ICNFG. The following procedure illustrates this link. The first thing to do is to move the contents of the @ICNFG vector into your initialization routine:

```
        LD      A,@FLAGS        ;Get flags pointer
        RST     28H             ;into register IY
        LD      A,(IY+28)       ;Get opcode
        LD      (LINK),A
        LD      L,(IY+29)       ;Get address LOW
        LD      H,(IY+30)       ;Get address HIGH
        LD      (LINK+1),HL
```

This subroutine does this by transferring the 3-byte vector to your routine. You then need to relocate your routine to its execution memory address. Once this is done, transfer the relocated initialization entry point to the @ICNFG vector as a jump instruction:

```
        LD      HL,INIT         ;Get (relocated)
        LD      (IY+29),L       ;init address
        LD      (IY+30),H
        LD      A,0C3H          ;Set JP instruction
        LD      (IY+28),A
```

If you need to invoke the initialization routine at this point, then you can use:

```
        CALL    ROUTINE         ;Invoke your routine
```

Your initialization routine would be unique to the function it was to perform, but an overall design would look like this:

```
INIT    CALL    ROUTINE         ;Start of init
LINK    DEFS    3               ;Continue on
ROUTINE .
        your initialization routine

RET
```

After linking in your routine, perform the SYSGEN. If you have followed these procedures, your routine will be invoked every time you start up TRSDOS.

Interfacing to @KITSK

Background tasks can be invoked in one of two ways. For tasks that do not require disk I/O, you can use the RTC (Real Time Clock) interrupt and one of the 12 task slots (or other external interrupt). For tasks that require disk I/O, you can use the keyboard task process.

At the beginning of the TRSDOS keyboard driver is a call to @KITSK. This means that any time that @KBD is called, the @KITSK vector is also called. (The type-ahead task, however, bypasses this entry so that @KITSK is not called from the type-ahead routine.) Therefore, if you want to interface a background routine that does disk I/O, you must chain into @KITSK.

The interfacing procedure to @KITSK is identical to that shown in the section "Interfacing to @ICNFG," except that IY+31 through IY+33 is used to reference the @KITSK vector. You may want to start your background routine with:

```
START   CALL    ROUTINE         ;Invoke task
LINK    DEFS    3               ;For @KITSK hook
ROUTINE EQU     $               ;Start of the task
```

Be aware of one major pitfall. The @KBD routine is invoked from @CMNDI and @CMNDR (which is in SYS1/SYS). This invocation is from the @KEYIN call, which fetches the next command line after issuing the "TRSDOS Ready" message. If your background task executes and opens or closes a file (or does anything to cause the execution of a system overlay other than SYS1), then SYS1 is overwritten by SYS2 or SYS3. When your routine finishes, the @KEYIN handler tries to return to what called it—SYS1, which is no longer resident. Therefore, any task chained to @KITSK which causes a resident SYS1 to be overwritten must reload SYS1 before returning.

You can use the following code to reload SYS1 if SYS1 was resident prior to your task's execution:

```
ROUTINE LD      A,@FLAGS        ;Get flags pointer
        RST     28H             ;into register IY
        LD      A,(IY-1)        ;Get resident over-
        AND     8FH             ;lay and remove
        LD      (OLDSYS+1),A    ;the entry code
        .
```

```
          rest of your task
          .
EXIT      EQU   $
OLDSYS    LD    A,0            ;Get old overlay #
          CP    83H            ;Was it SYS1?
          RET   NZ             ;Return if not; else
          RST   28H            ;Get SYS1 per reg. A
                               ;(no RET needed)
```

Interfacing to the Task Processor

This section explains how to integrate interrupt tasks into your applications.

One of the hardware interrupts in the TRS-80 is the real time clock (RTC). The RTC is synchronized to the AC line frequency and pulses at 60 pulses per second, or once every 16.67 milliseconds. (Computers operating with 50 Hz AC use a 50 pulses per second RTC interrupt. In this case, all time relationships discussed in this section should be adjusted to the 50 Hz base.)

A software task processor manages the RTC interrupt in performing background tasks necessary to specific functions of TRSDOS (such as the time clock, blinking cursor, and so on). The task processor allows up to 12 individual tasks to be performed on a "time-sharing" basis.

These tasks are assigned to "task slots" numbered from 0 to 11. Slots 0-7 are considered "low priority" tasks (executing every 266.67 milliseconds). Slots 8-10 are medium priority tasks (executing every 33.33 milliseconds). Slot 11 is a high priority task (executing every 16.66 milliseconds SYSTEM (FAST) or 33.33 milliseconds SYSTEM (SLOW)). Task slots 3, 7, 9, and 10 are reserved by the system for the ALIVE, TRACE, SPOOL, and TYPE-AHEAD functions, respectively.

TRSDOS maintains a Task Control Block Vector Table (TCBVT) which contains 12 vectors, one for each of the 12 task slots. TRSDOS contains five supervisor calls that manage the task vectors. The five SVCs and their functions are:

@CKTSK	Checks to see whether a task slot is unused or active
@ADTSK	Adds a task to the TCBVT
@RMTSK	Removes a task from the TCBVT
@KLTSK	Removes the currently executing task
@RPTSK	Replaces the TCB address for the current task

The TRSDOS Task Control Block Vector Table contains vector pointers. Each TCBVT vector points to an address in memory, which in turn contains the address of the task. Thus, the tasks themselves are indirectly addressed.

When you are programming a task to be called by the task processor, the entry point of the routine needs to be stored in memory. If you make this storage location the beginning of a Task Control Block (TCB), the reason for indirect vectoring of interrupt tasks will become more clear. Consider an example TCB:

```
MYTCB    DEFW  MYTASK
COUNTER  DEFB  15
TEMPY    DEFS  1
MYTASK   RET
```

This is a useless task, since the only thing it does is return from the interrupt. However, note that a TCB location has been defined as "MYTCB" and that this location contains the address of the task. A few more data bytes immediately following the task address storage have also been defined.

Upon entry to a service routine, index register IX contains the address of the TCB. You can therefore address any TCB data using index instructions. For example, you could use the instruction "DEC (IX+2)" to decrement the value contained in COUNTER in the above routine.

Here is the routine expanded slightly:

```
MYTCB    DEFW  MYTASK
COUNTER  DEFB  15
TEMPY    DEFB  0
MYTASK   DEC   (IX+2)
         RET   NZ
         LD    (IX+2),15
         RET
```

This version makes use of the counter. Each time the task executes, the counter is decremented. When the count reaches zero, the counter is restored to its original value.

In order to be executed, all tasks must be added to the TCBVT. The @ADTSK supervisor call does this. For the above routine, assume the task slot chosen is low-priority slot 2. You can ascertain that slot 2 is available for use by using the @CKTSK SVC as follows:

```
LD    C,2          ;Reference slot 2
LD    A,28         ;Set for @CKTSK SVC
RST   28H          ;An "NZ" indication
JP    NZ,INUSE     ;says that the slot is
                   ;being used.
```

Once you determine that the slot is available (that is, not being used by some other task), you can add your task routine. The following code adds this task to the TCBVT:

```
LD    DE,MYTCB     ;Point to the TCB
LD    C,2          ;Reference slot 2
LD    A,29         ;Set for @ADTSK SVC
RST   28H          ;Issue the SVC
```

The above program lines point register DE to the TCB, load the task slot number into register C, and then issue the @ADTSK supervisor call. If you want this task to run regardless of what is in memory, you can place it in high memory (of bank 0) and protect it by moving HIGH$ below it via the @HIGH$ supervisor call.

Once a task has been activated, it is sometimes necessary to deactivate it. You can do this in two ways. The most common way is to use the @RMTSK supervisor call:

```
LD    C,2          ;Designate the task
                   ;slot
LD    A,30         ;Set for @RMTSK SVC
RST   28H          ;Issue the SVC
```

You identify the task slot to remove by placing a value in register C, and then you issue the supervisor call.

You can use another method if you want to remove the task while it is being executed. Examine the routine modified as follows:

```
MYTCB    DEFW  MYTASK
COUNTER  DEFB  10
TEMPY    DEFB  0
MYTASK   DEC   (IX+2)
         RET   NZ
         LD    A,32         ;Set for @KLTSK SVC
         RST   28H          ;Issue the SVC
```

The @KLTSK supervisor call removes the currently executing task from the TCBVT. The system does not return to your routine, but continues as if you had executed a RET instruction. For this reason, the @KLTSK SVC should be the last instruction you want executed. In this example, MYTASK decrements the counter by one on each entry to the task. When the counter reaches zero, the task is removed from slot 2.

The last task processor supervisor call is @RPTSK. The @RPTSK function updates the TCB storage vector (the vector address in your Task Control Block) to be the address immediately following the @RPTSK SVC instruction. As with @KLTSK, the system does not return to your service routine after the SVC is made, but continues on with the task processor. The following example illustrates how @RPTSK can be used in a program:

```
         ORG    9000H
@ADTSK   EQU    29
@RPTSK   EQU    31
@RMTSK   EQU    30
@EXIT    EQU    22
@VDCTL   EQU    15
BEGIN    LD     DE,TCB         ;Point to TCB
         LD     C,0            ;and add the task
         LD     A,@ADTSK       ;to slot 0
         RST    28H
         LD     A,@EXIT        ;Exit to TRSDOS
         RST    28H
TCB      DEFW   TASK
COUNTER  DEFB   15
TASKA    LD     A,@RPTSK       ;Replace current
         RST    28H            ;task with TASKA
TASK     LD     BC,027CH       ;Put a character
         LD     HL,004FH       ;at Row 0, Col. 79
         LD     A,@VDCTL
         RST    28H
         DEC    (IX+2)         ;Decrement the counter
         RET    NZ             ;and return if not
         LD     (IX+2),15      ;expired; else reset
         LD     A,@RPTSK       ;Replace the previous
         RST    28H            ;task with TASKB
TASKB    LD     BC,022DH       ;Put a character
         LD     HL,004FH       ;at Row 0, Col. 79
         LD     A,@VDCTL
         RST    28H
         DEC    (IX+2)
         RET    NZ
         LD     (IX+2),15
         JR     TASKA
         END    BEGIN
```

This task routine contains no method of relocating it to protected RAM. The statements starting at the label BEGIN add the task to TCBVT slot 0 and return to TRSDOS Ready. The task contains a four-second down counter and a routine to put a character in video RAM (80th character of Row 0). At four-second intervals, the character toggles between '¦' and '–'. This is done by using the @RPTSK SVC to toggle the execution of two separate routines which perform the character display.

TRSDOS uses bank-switched memory. In order to properly control and manage this additional memory, certain restrictions are placed on tasks. All tasks must be placed either in low memory (addresses X'0000' through X'7FFF') or in bank zero of high memory (addresses X'8000' through X'FFFF'). The task processor always enables bank zero when performing background tasks. The assembly language programmer must ensure that tasks are placed in the correct memory area.

Interfacing RAM Banks 1 and 2

The proper use of the RAM bank transfer techniques described here requires a high degree of skill in assembly language programming. This section on bank switching is intended for the professional.

The TRS-80 Model 4 can optionally support a second set of 64K RAM, bringing the total RAM to 128K. TRSDOS designates this extra 64K RAM as two banks of 32K RAM each, which are banks 1 and 2 of bank-switched RAM. The upper 32K of standard RAM is designated bank 0. At any one time, only one of the banks is resident. The resident bank is always addressed at X'8000' through X'FFFF.' When a bank transfer is performed, the specified bank becomes addressable and the previous bank is no longer available. Since memory refresh is performed on all banks at all times, nothing in the previously resident bank is altered during whatever time it is not addressable (that is, not resident).

You can access this additional RAM by means of the @BANK supervisor call (SVC 102). When you power up your computer or press reset, TRSDOS looks to see which banks of RAM are installed in your machine. TRSDOS maintains a bit map in one byte of storage, with each bit representing one of the banks of RAM. This byte is called "Bank Available RAM" (BAR), and its information is set when you boot TRSDOS. Bit 0 corresponds to bank 0, bit 1 corresponds to bank 1, and so on up to bit 7. From a hardware standpoint, the Model 4 has a maximum of three banks. You have either bank 0 only (a 64K machine), or banks 0-2 (a 128K machine).

Another bit map is used to indicate whether a bank is reserved or available for use. This byte is called the "Bank Used RAM" (BUR). Again, bit 0 corresponds to bank 0, bit 1 to bank 1, and so on. TRSDOS design supports the use of banks 1 and 2 primarily for data storage (for example, a spool buffer, Memdisk, etc.). The management of any memory space within a particular bank of RAM (excluding bank 0) is the responsibility of the application program "reserving" a particular bank.

TRSDOS requires that any device driver or filter that is relocated to high memory (X'8000' through X'FFFF') reside in bank 0. The TRSDOS device handler always invokes bank 0 upon execution of any byte I/O service request (@PUT, @GET, @CTL, as well as other byte I/O SVCs that use @PUT/@GET/@CTL). This ensures that any filter or driver attached to the device in question will be available. If a RAM bank other than 0 was resident, it is restored upon return from the device handler. This ensures that device I/O is never impacted by bank switching.

TRSDOS also requires that all interrupt tasks reside in bank 0 or low memory (X'0000' through X'7FFF'). The interrupt task processor always enables bank 0 and restores whatever bank was previously resident. An interrupt task may perform a bank transfer from 0 to another bank provided the necessary linkage and stack area is used. This is discussed in more detail later.

All bank transfer requests must be performed using the @BANK SVC. This SVC provides four functions, three of which are interrogatory and one of which performs the actual bank switching.

As mentioned previously, the contents of banks other than 0 are managed by the application, not by TRSDOS. Therefore, the application needs a way of finding out if any given bank is available. For example, if an application wants to reserve use of bank 1, it must first check to see if bank 1 is free to use. This is done by using function 2 as follows:

```
LD      C,1           ;Specify bank 1
LD      B,2           ;Check BUR if bank in use
LD      A,@BANK       ;Set @BANK SVC (102)
RST     28H
JR      NZ,INUSE      ;NZ if bank already in use
```

Note that the return condition (NZ or Z) shows whether or not you can use the specified bank (it may not even be installed).

If the specified bank is available, you then need to reserve it. Do this by using function 3 as follows:

```
LD      C,1           ;Specify bank 1
LD      B,3           ;Set BUR to show "in use"
```

```
LD      A,@BANK        ;Set @BANK SVC (102)
RST     28H
JR      NZ,ERROR
```

You must check for an error by examining the Z flag. In general (discounting a system error), an NZ condition returned means that the specified bank is already in use. If you had performed a function 2 (testing to see if the bank was available) and got a not-in-use indication, but got an NZ condition on function 3, then the @BANK SVC routine has been altered and is probably unusable.

When an application no longer requires a memory bank, it can return the bank to a "free" state by using function 1 as follows:

```
LD      C,1            ;Specify bank 1
LD      B,1            ;Set BUR to show free
LD      A,@BANK        ;Set @BANK SVC (102)
RST     28H
```

No error condition is checked, as none is returned by TRSDOS. If you should mistakenly use function 1 with a bank that is nonexistent, an error is returned if you try to invoke the nonexistent bank.

To find out which bank is resident at any time, use function 4 as follows:

```
LD      B,4            ;Which bank is resident?
LD      A,@BANK        ;Set @BANK SVC (102)
RST     28H
```

The current bank number is returned in register A.

To exchange the current bank with the specified bank, use function 0. Since a memory transfer takes place in the address range X'8000' through X'FFFF,' the transfer cannot proceed correctly if the stack pointer (SP) contains a value that places the stack in that range. @BANK inhibits function 0 and returns an SVC error if the stack pointer violates this condition.

A bank can be used purely as a data storage buffer. The application's routines for invoking and indexing the bank switching probably reside in the user range X'3000' through X'7FFF.' As an example, the following code invokes a previously tested and reserved bank (via functions 2 and 3), accesses the buffer, and then restores the previous bank:

```
LD      C,1            ;Specify bank 1
LD      B,0            ;Bring up bank
LD      A,@BANK        ;Set @BANK SVC (102)
RST     28H
JR      NZ,ERROR       ;Error trap
PUSH    BC             ;Save old bank data
.
your code to access the buffer region
.
POP     BC             ;Recover old bank data
LD      A,@BANK        ;Set @BANK SVC (102)
RST     28H
JR      NZ,ERROR       ;Error trap
```

Note that the @BANK function 0 conveniently returns a zero in register B to effect a function 0 later, as well as provides the old bank number in register C. This means that you only have to save register pair BC, pop it when you want to restore the previous bank, and then issue the @BANK SVC.

Suppose you want to transfer to another bank from a routine that is executing in high memory. (Recall that the only limitation is that the stack must not be in high memory.) The @BANK SVC function 0 provides a technique for automatically transferring to an address in the new bank. This technique is called the transfer function. It relies on the assumption that since you are managing the *entire* 32K bank 1 or 2, your application should know exactly where it needs to transfer (that is, where the application originally placed the code to execute).

The code to perform a bank transfer is similar to the above example. Register pair HL is loaded with the transfer address. Register C, which contains the number of the bank to invoke, must have its high order bit (bit 7) set. After the specified bank is enabled, control is passed to the transfer address that is in HL. Upon entry to your routine in the new bank (referred to here as "PROGB"), register HL will contain the old return address so that PROGB will know where to return transfer. Register C will also contain the old bank number with bit 7 set and register B will contain a zero. This register set-up provides for an easy return to the routine in the old bank that invoked the bank transfer. An illustration of the transfer code follows:

```
        LD      C,1             ;Specify bank 1
        LD      B,0             ;Bring up bank 0
        LD      HL,(TRAADR)     ;Set the transfer
                                ;address
        SET     7,C             ;and denote a
                                ;transfer
        LD      A,@BANK         ;Set @BANK SVC (102)
        RST     28H
RETADR  JR      NZ,ERROR
```

Control is returned to "RETADR" under either of two conditions. If there was an error in executing the bank transfer (for example, if an invalid bank number was specified or the stack pointer is in high memory), the returned condition is NZ. If the transfer took place and PROGB transferred back, the returned condition is Z. Thus, the Z flag shows whether or not there was a problem with the transfer.

If PROGB needs to provide a return code, it must be done by using register pair DE, IX, or IY, as registers AF, BC, and HL are used to perform the transfer. (Or, some other technique can be used, such as altering the return transfer address to a known error trapping routine.)

PROGB should contain code that is similar to that shown earlier. For example, PROGB could be:

```
PROGB   PUSH    BC              ;Save old bank data
        PUSH    HL              ;Save the RET
                                ;address
        .
        your PROGB routines
        .
        POP     HL              ;Recover transfer
                                ;address
        POP     BC              ;Get bank transfer
                                ;data
        LD      A,102           ;Set @BANK SVC
        RST     28H
        JR      NZ,ERROR        ;Error trap
```

PROGB saves the bank data (register BC). Don't forget that a transfer was effected and register C has bit 7 already set when PROGB is entered. PROGB also saves the address it needs to transfer back (which is in HL). It then performs whatever routines it has been coded for, recovers the transfer data, and issues the bank transfer request. As explained earlier, an NZ return condition from the @BANK SVC indicates that the bank transfer was not performed. You should verify that your application has not violated the integrity of the stack where the transfer data was stored.

Never place disk drivers, device drivers, device filters, or interrupt service routines in banks other than bank 0. It is possible to segment one of the above modules and place segments in bank 1 or 2, provided the segment containing the primary entry is placed in bank 0. You can transfer between segments by using the bank transfer techniques discussed above.

Device independence has its roots in "byte I/O." Byte I/O is any I/O passed through a device channel one byte at a time.

Three primitive routines are available at the assembly language level for byte I/O. These byte I/O primitives can be used to build larger routines. The three primitives are the TRSDOS supervisor calls @GET, @PUT, and @CTL. @GET is used to input a byte from a device or file. @PUT is used to output a byte to a device or file. @CTL is used to communicate with the driver routine servicing the device or file.

Other supervisor calls perform byte I/O, such as @KBD (scan the keyboard and return the key code if a key is down), @DSP (display a character on the video screen), and @PRT (output a character to the line printer). These functions operate by first loading register pair DE with a pointer to a specific Device Control Block (DCB) assigned for use by the device, then issuing a @GET or @PUT SVC for input or output requests.

When TRSDOS passes control over to the device driver routine, the Z-80 flag conditions are unique for each different primitive. This enables the driver to establish which primitive was used to access the routine, so it can turn over the I/O request to the proper driver or filter subroutine according to the type of request — input, output, or control.

The following table shows the FLAG register conditions upon entry to a driver or filter:

C,NZ = @GET primitive
Z,NC = @PUT primitive
NZ,NC = @CTL primitive

Register B contains the I/O direction code: 1 = @GET, 2 = @PUT, 4 = @CTL. Register C contains the character code that was passed in the @PUT or @CTL supervisor call. Register IX points to the TYPE byte (DCB + 0) of the Device Control Block. Registers BC, DE, HL, and IX have been saved on the stack and are available for use. Register AF is not saved; if you want it preserved, your program must do so.

Your driver must start with a standard front-end header (see "Memory Header"):

```
BEGIN     JR      START          ;Go to actual code
                                 ;beginning
          DEFW    MODEND-1       ;Last byte used by
                                 ;module
          DEFB    7              ;Length of name
          DEFM    'MODNAME'      ;Name
MODDCB    DEFW    $-$            ;DCB ptr. for this
                                 ;module
          DEFW    0              ;Reserved by TRSDOS
```

At the start of the actual module code, test the condition of the F register flags for @GET, @PUT, and @CTL:

```
START     EQU     $
;         Actual module code start
          JR      C,WASGET       ;Go if @GET request
          JR      Z,WASPUT       ;Go if @PUT request
          .                      ;Was @CTL request
```

At the label START, a test is made on the carry flag. If the carry was set, then the disk primitive must have been an input request (@GET). An input request could be directed to a part of the driver which only handles input from the device.

If the request was not from the @GET primitive, the carry will not be set. The next test checks to see if the zero flag is set. The zero condition is preset when a @PUT primitive was the initial request. The jump to WASPUT can go to a part of the driver that deals specifically with output to the device.

If neither the zero nor carry flags are set, the routine falls through to the next instruction (not shown), which would begin the part of the driver that handles @CTL calls. For example, you may want to have an RS-232C driver handle a BREAK by issuing a @CTL call so that the RS-232C driver emits a true modem break, but a CONTROL C would @PUT a X'03'.

Some drivers are written to assume that @CTL requests are to be handled exactly like @PUT requests. This is entirely up to the author and the function of the driver.

Note that when a device is routed to a disk file, TRSDOS ignores @CTL requests. That is, the @CTL codes are not written to the disk file.

On @GET requests, the character input should be placed in the accumulator. On output requests (either @PUT or @CTL), the character is obtained from register C. It is important for drivers and filters to observe return codes. Specifically, if the request is @GET and no byte is available, the driver returns an NZ condition and the accumulator contains a zero (that is, OR 1 : LD A,0 : RET). If a byte is available, the byte is placed in the accumulator and the Z flag is set (that is, LD A,CHAR : CP A : RET). If there is an input error, the error code is returned in the accumulator and the Z flag is reset (that is, LD A,ERRNUM : OR A : RET). On output requests, the accumulator will contain the byte output with the Z flag set if no error occurred. In the case of an output error, the accumulator must be loaded with the error code and the Z flag reset as shown above.

A filter module is inserted between the DCB and driver routine (or between the DCB and the current filter when it is applied to a DCB already filtered). The insertion is performed by the TRSDOS FILTER command once the filter module is resident and attached to a phantom DCB. The usual linkage for a filter is to access the chained module by calling the @CHNIO supervisor call with specific linkage data in registers IX and BC. Register IX is loaded with the filter's DCB pointer obtained from the memory header MODDCB pointer. Register B must contain the I/O direction code (1 = @GET, 2 = @PUT, 4 = @CTL). This code is already in register B when the filter is entered. You can either keep register B undisturbed or load it with the proper direction code. Also, output requests expect the output byte to be in register C.

The DCB pointer obtained from MODDCB is passed in register DE by the SET command and is loaded into MODDCB by your filter initialization routine. The initialization routine needs to relocate the filter to high memory and attach itself to the DCB assigned by the SET command. If the initialization front end had transferred the DCB pointer from DE to IX, then the following code could be used to establish the TYPE byte and vector for the filter:

```
LD      (IX),47H        ;Init DCB type to
LD      (IX+1),E        ;FILTER, G/P/C I/O,
LD      (IX+2),D        ;& stuff vector
```

A filter module can operate on input, output, control, or any combination based on the author's design. The memory header provides a region for user data storage conveniently indexed by the module.

An illustration of a filter follows. The purpose of this filter is to add a linefeed on output whenever a carriage return is to be sent. Although the filter requires no data storage, the technique for accessing data storage is shown.

```
BEGIN    JR      START          ;Branch to start
         DEFW    FLTEND-1       ;Last byte used
         DEFB    6              ;Name length
         DEFM    'SAMPLE'       ;Name
MODDCB   DEFW    0              ;Link to DCB
         DEFW    0              ;Reserved
;        Data storage area for your filter
CR       EQU     0DH
LF       EQU     0AH
DATA$    EQU     $
DATA1    EQU     $-DATA$
         DEFB    0              ;Data storage
DATA2    EQU     $-DATA$
         DEFB    0              ;Data storage
;        Start of filter
START    JR      Z,GOTPUT       ;Go if @PUT
;        @GET and @CTL requests are chained to
;        the next module attached to the device.
;        This is accomplished by falling through
;        to the @CHNIO call. Note that the sample
;        filter does not affect the B register,
;        so the filter does not have to load it
;        with the direction code.
FLTPUT   PUSH    IX             ;Save your data
                                ;pointer
         LD      IX,(MODDCB)
RX01     EQU     $-2            ;Grab the DCB vector
         LD      A,@CHNIO       ;and chain to it
         RST     28H
         POP     IX
         RET
;        Filter code
GOTPUT   LD      IX,PFDATA$     ;Base register is
RX02     EQU     $-2            ;used to index data
         LD      A,C            ;Get character to
                                ;test
         CP      CR             ;If not CR, put it
         JR      NZ,FLTPUT
         CALL    FLTPUT         ;else put it
RX03     EQU     $-2
         RET     NZ             ;Back on error
         LD      C,LF           ;Add linefeed
         JR      FLTPUT
FLTEND   EQU     $
;        Relocation table
RELTAB   DEFW    RX01,RX02,RX03
TABLEN   EQU     $-RELTAB/2
```

The relocation table, RELTAB, would be used by the filter initialization relocation routine.

@CTL Interfacing to Device Drivers

This section discusses the @CTL functions supported by the system device drivers. To invoke a @CTL function, point register pair DE to the Device Control Block (DCB), load the function code into register C, and issue the @CTL supervisor call. You can locate the DCB address by either 1) using the @GTDCB SVC, or 2) using the @OPEN SVC to open a File Control Block containing the device specification and using the FCB address. See the @CTL supervisor call for a list of the function codes and their meanings.

The @CTL functions are listed below for each driver.

Keyboard Driver (resident driver assigned to *KI)

A function value of X'03' clears the type-ahead buffer. This serves the same purpose as repeated calls to @KBD until no character is available.

A function value of X'FF' is reserved for system use.

All other function values are treated as @GET requests.

The module name assigned to this driver is "$KI".

Video Driver (resident driver assigned to *DO)

All @CTL requests are treated as if they were @PUT requests.

The module name assigned to this driver is "$DO".

Printer Driver (resident driver assigned to *PR)

The printer driver is transparent to all code values when requested by the @PUT SVC. That means that all values from X'00' through X'FF' (0-255) can be sent to the printer. If the FORMS filter is attached to the *PR device, then various codes are trapped and used by the filter according to parameters specified with the FORMS library command, as follows:

X'0D' —Generates a carriage return and optionally a linefeed (ADDLF). Generates form feeds as required.
X'0A' —Treated the same way as X'0D'.
X'0C' —Generates form feeds (via repeated line feeds if soft form feed). (FFHARD=OFF)
X'09' —Advances to next tab column.
X'06' —Sets top-of-form by resetting the internal line counter to zero.

Other character codes may be altered if the user translation option of the FORMS command (XLATE) is set.

The printer driver accepts a function value of X'00' via the @CTL request to return the printer status. If the printer is available, the Z flag will be set and register A will contain X'30'. If the Z flag is reset, register A will contain the four high-order bits of the parallel printer port (bits 4-7).

The module name assigned to the printer driver is "$PR". The module name of the FORMS filter is "$FF".

COM Driver (non-resident driver for the RS-232C)

This driver handles the interfacing between the RS-232C hardware and byte I/O (usually the *CL device).

A @CTL function value of X'00' returns an image of the RS-232 status register in the accumulator. The Z flag will be set if the RS-232 is available for "sending" (that is, if the transmit holding register is empty and the flag conditions match as specified by SETCOM).

A function value of X'01' transmits a "modem break" until the next character is @PUT to the driver.

A function value of X'02' re-initializes the UART to the values last established by SETCOM.

A function value of X'04' enables or disables the WAKEUP feature.

All other function values are ignored and the driver returns with register A containing a zero value and the Z flag set.

The WAKEUP feature is useful for application software specializing in communications. The RS-232 hardware can generate a machine interrupt under any of three conditions: when the transmit holding register is empty, when a received character is available, or when an error condition has been detected (framing error, parity error, and so on). The COM driver makes use of the "received character available" interrupt to take control when a fully formed character is in the holding register. The COM driver services the interrupt by reading the character and storing it in a one-character buffer. COM then normally returns from the interrupt.

An application can request that, instead of returning, control be passed to the application for immediate attention. Note that this action would occur during interrupt handling, and any processing by the application must be kept to a minimum before control is returned to COM via a RET instruction.

If you use a @CTL function value of X'04,' then register IY must contain the address of the handling routine in your application. Upon return from the @CTL request, register IY contains the address of the previous WAKEUP vector. This should be restored when your application is finished with the WAKEUP feature.

When control is passed to your WAKEUP vector upon detection of a "receive character available" interrupt, certain information is immediately available. Register A contains an image of the UART status register. The Z flag is set if a valid character is actually available. The character, if any, is in the C register.

Since system overhead takes a small amount of time in the @GET supervisor call, you may need to @GET the character via standard device interfacing. This ensures that any filtering or linking in the *CL device chain will be honored. If, on the other hand, your application is attempting to transfer data at a very high rate (9600 baud or higher), you may need to bypass the @GET SVC and use the character immediately available in the C register. Note that this procedure bypasses the normal device chain (device routing and linking).

The module name of the COM driver is "$CL."

8/Using the Supervisor Calls

Supervisor Calls (SVCs) are operating system routines that are available to assembly language programs. These routines alter certain system functions and conditions, provide file access, and perform various computations. They also perform I/O to the keyboard, video display, and printer.

Each SVC has a number which you specify to invoke it. These numbers range from 0 to 104.

In addition, under Version 6.2, you can write your own operating system routines using the numbers 124 through 127 to install your own SVC's. See Appendix E, "Programmable SVCs" for more information.

Calling Procedure

To call a TRSDOS SVC:

1. Load the SVC number for the desired SVC into register A. Also load any other registers which are needed by the SVC, as detailed under Supervisor Calls.
2. Execute a RST 28H instruction.

Note: If the SVC number supplied in register A is invalid, the system prints the message "System Error *xx*", where *xx* is usually 2B. It then returns you to TRSDOS Ready (*not* to the program that made the invalid SVC call).

The alternate register set (AF', BC', DE', HL') is not used by the operating system.

Program Entry and Return Conditions

When a program executed from the @CMNDI SVC is entered, the system return address is placed on the top of the stack. Register HL will point to the first non-blank character following the command name. Register BC will point to the first byte of the command line buffer.

Three methods of return from a program back to the system are available: the @ABORT SVC, the @EXIT SVC, and the RET instruction. For application programs and utilities, the normal return method is the @EXIT SVC. If no error condition is to be passed back, the HL register pair must contain a zero value. Any non-zero value in HL causes an active JCL to abort.

The @ABORT SVC can be used as an error return back to the system; it automatically aborts any active JCL processing. This is done by loading the value X'FFFF' into the HL register pair and internally executing an @EXIT SVC.

If stack integrity is maintained, a RET instruction can be used since the system return address is put on the stack by @CMNDI. This allows a return if the program was called with @CMNDR.

Most of the SVCs in TRSDOS Version 6 set the Z flag when the operation specified was successful. When an operation fails or encounters an error, the Z flag is reset (also known as NZ flag set) and a TRSDOS error code is placed in the A register. The remaining SVCs use the Z/NZ flag in differing ways, so you should refer to the description of the SVCs you are using to determine the exit conditions.

Supervisor Calls

The TRSDOS Supervisor Calls are:

Keyboard SVCs

@CKBRKC
@KBD
@KEY
@KEYIN

Printer and Video SVCs

@CLS
@DSP
@DSPLY
@LOGER
@LOGOT
@MSG
@PRT
@PRINT
@VDCTL

Disk SVCs

@DCINIT
@DCRES
@DCSTAT
@RDSEC
@RDSSC
@RSLCT
@RSTOR
@SEEK
@SLCT
@STEPI
@VRSEC
@WRSEC
@WRSSC
@WRTRK

System Control SVCs

@ABORT
@BREAK
@CMNDI
@CMNDR
@EXIT
@FLAGS
@HIGH$
@IPL
@LOAD
@RUN

Special Purpose Disk SVCs

@DIRRD
@DIRWR
@GTDCT
@HDFMT
@RDHDR
@RDTRK

Byte I/O SVCs

@CTL
@GET
@PUT

File Control SVCs

@CLOSE
@FEXT
@FNAME
@FSPEC
@INIT
@REMOV
@OPEN
@RENAM

Disk File Handler SVCs

@BKSP
@CKEOF
@LOC
@LOF
@PEOF
@POSN
@READ
@REW
@RREAD
@RWRIT
@SEEKSC
@SKIP
@VER
@WEOF
@WRITE

TRSDOS Task Control SVCs

@ADTSK
@CKTSK
@KLTSK
@RMTSK
@RPTSK

Special Overlay SVCs

@CKDRV
@DEBUG
@DODIR
@ERROR
@PARAM
@RAMDIR

Miscellaneous SVCs

@BANK
@DATE
@DECHEX
@DIV8
@DIV16
@HEXDEC
@HEX8
@HEX16
@MUL8
@MUL16
@PAUSE
@SOUND
@TIME
@WHERE

Special Purpose SVCs

@CHNIO
@GTDCB
@GTMOD

See the pages that follow for a detailed description of each supervisor call.

Abort Program

Loads HL with an X'FFFF' error code and exits through the @EXIT supervisor call. Any active JCL processing is aborted.

Entry Conditions:

A=21 (X'15')

General:

This SVC does not return.

Example:

See the example for @EXIT in Sample Program B, lines 206-207.

Add an Interrupt Level Task

Adds an interrupt level task to the real time clock task table. The task slot number can be 0-11; however, some slots are already assigned to certain functions in TRSDOS. Slot assignments 0-7 are low priority tasks executing every 266.67 milliseconds. Slots 8-10 are medium priority tasks executing every 33.33 milliseconds. Slot 11 is a high priority task, executing every 16.66 milliseconds High Speed or 33.33 milliseconds Low Speed. The system uses task slots 3, 7, 9, and 10 for the ALIVE, TRACE, SPOOL, and TYPE-AHEAD functions, respectively.

It is a good practice to remove an existing task (using the @RMTSK or @KLTSK SVC) before installing a new task in the same task slot.

Entry Conditions:

A = 29 (X'1D')
DE = *pointer to Task Control Block (TCB)*
C = *task slot assignment* (0-11)

Exit Conditions:

Success always.
HL and AF are altered by this SVC.

The Task Control Block, or TCB, is a 2-byte block of RAM which contains the address of the task driver entry point. If your task is prefixed with the memory header described earlier under "Device Access," then the TCB can be stored in the memory header data storage area. If the task is not a driver or filter, the TCB can be stored in the memory header location MODDCB. Upon entry to your task routine, the IX register contains the TCB address.

Example:

See Sample Program F, lines 109-120.

Memory Bank Use

Controls 32K memory bank operation. The top half of the main 64K block is bank 0, and the alternate 64K block is divided into banks 1 and 2. The system maintains two locations to perform bank management. These areas are known as "bank available RAM" (BAR) and "bank in use RAM" (BUR).

If the Stack Pointer is not X'7FFE' or lower, the SVC aborts with an Error 43 only if B=0.

Entry Conditions:

A= 102 (X'66')
B selects one of the following functions:
 If B=0, the specified bank is selected and is made addressable. The 32K bank starts at X'8000' and ends at X'FFFF'.
  C =*bank number to be selected* (0-2)
   If bit 7 is set, then execution will resume in the newly loaded bank at the address specified.
  HL=*address to start execution in the new bank*
 If B=1, reset BUR and show the bank not in use.
  C=*bank number to be selected* (0-2)
 If B=2, test BUR if bank is in use.
  C=*bank number to be selected* (0-2)
 If B=3, set BUR to show bank in use.
  C=*bank number to be selected* (0-2)
 If B=4, return number of bank currently selected.

Exit Conditions:

If B=0:
 Success, Z flag set.
  C =*the bank number that was replaced.* If bit 7 was set in register C on entry, it is also set on exit.
  HL=*SVC return address.* By keeping the contents of C and HL, you can later return to the instruction following the first @BANK SVC. See "Interfacing RAM Banks 1 and 2" for more information.
 Failure, NZ flag set. Bank not present or parameter error.
  A =*error number*

If B=1:
 Success, Z flag set. Bank available for use.
 Failure, NZ flag set. Bank not present.

If B=2:
 Success always.
  If Z flag is set, then the bank is available for use.
  If NZ flag is set, then test register A:
   If A≠X'2B,' then the bank is either in use or it does not exist on your machine. Banks 1 and 2 produce this error on a 64K machine.
   If A=X'2B,' then an entry parameter is out of range.

If B=3:
 Success, Z flag set. Bank is now reserved for your use.
 Failure, NZ flag set. Test register A:
  If A ≠ X'2B,' then the bank is already in use or does not exist. Banks 1 and 2 produce this error on a 64K machine.
  If A=X'2B,' then an entry parameter is out of range.

If B = 4:

Success always.

A = *number of the bank which is currently resident*

General:

AF is altered for all functions.

BC is altered if the SVC is successful.

Example:

See the section "Interfacing RAM Banks 1 and 2."

Backspace One Logical Record

Performs a backspace of one logical record.

Entry Conditions:
A = 61 (X'3D')
DE = *pointer to FCB of the file to backspace*

Exit Conditions:
If the Z flag is set or if A = X'1C' or X'1D', then the operation was successful. The LOC pointer to the file was backspaced one record. Otherwise, A = *error number*.
If A = X'1C' is returned, the file pointer is positioned at the end of the file. Any Appending operations would be performed here.
If A = X'1D' is returned, the file pointer is positioned beyond the end of the file.

General:
Only AF is altered by this SVC.
If the LOC pointer was at record 0 when the call was executed, the results are indeterminate.

Example:
See the example for @LOC in Sample Program C, lines 305-311.

Set Break Vector

Sets a user or system break vector. The BREAK vector is an abort mechanism; there is no return.

The BREAK vector executes whenever the following conditions occur at the same time: 1) the Program Counter is greater than X'2400', 2) the BREAK key is pressed, and 3) a real time clock interrupt which executes 30 times per second occurs.

After executing this SVC, you must reset bit 4 of SFLAG$. The BREAK flag in KFLAG$ (bit 0) requires the setting of SFLAG$ bit 4 and a delay of 0.1 to 0.5 second to clear any other interrupts that may be pending. Then you can enter your BREAK key handler (in which the BREAK key bit in SFLAG$ is reset). See KFLAG$ and SFLAG$ in the section about the @FLAGS SVC for more information.

Entry Conditions:
A = 103 (X'67')
HL = *user break vector*
HL = 0 (sets system break vector)

Exit Conditions:
Success always.
HL = *existing break vector* (if user break vector was set)

Note: @EXIT and @CMNDI automatically restore BREAK to the system handler. @CMNDR does not do this.

Pass Control to Next Module in Device Chain

Passes control to the next module in the device chain.

Entry Conditions:
A = 20 (X'14')
IX = *contents of DCB in the header block*
B = *GET/PUT/CTL direction code (1/2/4)*
C = *character* (if output request)

General:
IX is not checked for validity.

Example:
See the section "Device Driver and Filter Templates."

Check BREAK bit and clear it — Version 6.2 only

Checks to see if the BREAK key has been pressed. If a BREAK condition exists, @CKBRKC resets the break bit, Bit 0 of KFLAG$.

Entry Conditions:

A = 106(X'6A')

Exit Conditions:

Success always.
If Z flag is set, the break bit was not detected. If NZ flag is set, the break bit was detected and is cleared. If the BREAK key is being depressed, the SVC will not return until the key is released.

General:

Only AF is altered by this SVC.

Check Drive

Checks a drive reference to ensure that the drive is in the system and a TRSDOS Version 6 or LDOS 5.1.3 (Model III Hard Disk Operating System) formatted disk is in place.

Entry Conditions:

A=33 (X'21')
C=*logical drive number* (0-7)

Exit Conditions:

Success always.
If Z flag is set, the drive is ready.
If CF is set, the disk is write protected.
If NZ flag is set, the drive is not ready. The user may examine DCT+0 to see if the drive is disabled.

Example:

See Sample Program D, lines 35-55.

Check for End-Of-File

Checks for the end of file at the current logical record number.

Entry Conditions:
A = 62 (X'3E')
DE = *pointer to the FCB of the file to check*

Exit Conditions:
Success always.
If Z flag is set, LOC does not point at the end of file (LOC < LOF).
If NZ flag is set, test A for error number:
If A = X'1C,' LOC points at the end of the file (LOC = LOF).
If A = X'1D,' LOC points beyond the end of the file (LOC > LOF).
If A ≠ X'1C' or X'1D,' then A = *error number.*

General:
Only AF is altered by this SVC.

Example:
See Sample Program C, lines 352-353.

Check if Task Slot in Use

Checks to see if the specified task slot is in use.

Entry Conditions:
A = 28 (X'1C')
C = *task slot to check* (0-11)

Exit Conditions:
Success always.
If Z flag is set, the task slot is available for use.
If NZ flag is set, the task slot is already in use.

General:
AF and HL are altered by this SVC.

Example:
See Sample Program F, lines 70-73.

Close a File or Device

Terminates output to a file or device. Any unsaved data in the buffer area is saved to disk and the directory is updated. All files that have been written to must be closed, as well as all files opened with UPDATE or higher access.

If you remove a diskette containing an open file, any attempt to close the file results in the message:

> ** CLOSE FAULT ** *error message,* <ENTER> to retry, <BREAK> to abort

where *error message* is usually "Drive not ready" You may put the diskette back in the drive and:

1. Press (ENTER) to close the file.
2. Press (BREAK) to abort the close.

If you press (BREAK), the NZ flag is set and Register A contains X'20', the error code for an Illegal drive number error.

Entry Conditions:
A =60 (X'3C')
DE=*pointer to FCB or DCB to close*

Exit Conditions:
Success, Z flag set. The file or device was closed. The filespec (excluding the password) or the devspec is returned to the FCB or DCB.
Failure, NZ flag set.
A=*error number*

General:
Only AF is altered by this SVC.

Example:
See Sample Program C, lines 360-368.

Clear Video Screen — Version 6.2 only

Clears the video screen by sending a Home Cursor (X'1C') and Clear to End of Frame (X'1F') sequence to the video driver.

Entry Conditions:

A = 105(X'69')

Exit Conditions:

Success, Z flag is set.
Failure, NZ is set.
A = *error number*

General:

Only AF is altered by this SVC.

Execute Command with Return to System

Passes a command string to TRSDOS for execution. After execution is complete, control returns to TRSDOS Ready. If the command gets an error, it still returns to TRSDOS Ready.

Entry Conditions:
A = 24 (X'18')
HL = *pointer to buffer containing command string terminated with X'0D'* (up to 80 bytes, including the X'0D')

General:
This SVC does not return.

Example:
See Sample Program E, lines 43-58.

Execute Command

Executes a command or program and returns to the calling program. The executed program should maintain the Stack Pointer and exit via a RET instruction. All TRSDOS library commands comply with this requirement.

If bit 4 of CFLAG$ is set (see the @FLAGS SVC), then @CMNDR executes only system library commands.

Entry Conditions:
A =25 (X'19')
HL=*pointer to buffer containing command string terminated with X'ØD'* (up to 8Ø bytes, including the X'ØD')

Exit Conditions:
Success always.
HL=*return code* (See the section "Converting to TRSDOS Version 6" for information on return codes.)
Registers AF, BC, DE, IX, and IY are altered by the command or program executed by this SVC.
If the command invokes a user program which uses the alternate registers, they are modified also.

Example:
See Sample Program E, lines 18-29.

Output a Control Byte

Outputs a control byte to a logical device. The DCB TYPE byte (DCB + 0, Bit 2) must permit CTL operation. See the section "@CTL Interfacing to Device Drivers" for information on which of the functions listed below are supported by the system device drivers.

Entry Conditions:

A = 5 (X'05')
DE = *pointer to DCB to control output*
C selects one of the following functions:
If C = 0, the status of the specified device will be returned.
If C = 1, the driver is requested to send a BREAK or force an interrupt.
If C = 2, the initialization code of the driver is to be executed.
If C = 3, all buffers in the driver are to be reset. This causes all pending I/O to be cleared.
If C = 4, the wakeup vector for an interrupt-driven driver is specified by the caller.
IY = address to vector when leaving driver. If IY = 0, then the wakeup vector function is disabled. The RS-232C driver COM/DVR ($CL), is the only system driver that provides wakeup vectoring.
If C = 8, the next character to be read will be returned. This allows data to be "previewed" before the actual @GET returns the character.

Exit Conditions:

If C = 0,
Z flag set, device is ready
NZ flag set, device is busy
A = status image, if applicable
Note: This is a hardware dependent image.

If C = 1,
Success, Z flag set. BREAK or interrupt generated.
Failure, NZ flag set
A = *error number*

If C = 2,
Success, Z flag set. Driver initialized.
Failure, NZ flag set
A = *error number*

If C = 3,
Success, Z flag set. Buffers cleared.
Failure, NZ flag set.
A = *error number*

If C = 4,
Success always.
IY = *previous vector address*
This function is ignored if the driver does not support wakeup vectoring.

If C = 8,
Success, Z flag set. Next character returned.
A = *next character in buffer*
Failure, NZ flag set. Test register A:
If A = 0, no pending character is in buffer
If A ≠ 0, A contains *error number*. (TRSDOS driver returns Error 43.)

General:

BC, DE, HL, and IX are saved.

Function codes 5 to 7, 9 to 31, and 255 are reserved for the system. Function codes 32 to 254 are available for user definition.

Entry and exit conditions for user-defined functions are up to the design of the user-supplied driver.

Example:

See the section "Device Driver and Filter Templates."

Get Date

Returns today's date in display format (MM/DD/YY).

Entry Conditions:
A = 18 (X'12')
HL = *pointer to 8-byte buffer to receive date string*

Exit Conditions:
Success always.
HL = *pointer to the end of the buffer supplied + 1*
DE = *pointer to start of DATE$ storage area in TRSDOS*
BC is altered by this SVC.

Example:
See Sample Program F, lines 252-253.

Initialize the FDC

Issues a disk controller initialization command. The floppy disk driver treats this the same as @RSTOR (SVC 44).

Entry Conditions:
A = 42 (X'2A')
C = *logical drive number* (0-7)

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

Example:
See the example for @CKDRV in Sample Program D, lines 38-39.

Reset the FDC

Issues a disk controller reset command. The floppy disk driver treats this the same as @RSTOR (SVC 44).

Entry Conditions:
A = 43 (X'2B')
C = *logical drive number* (0-7)

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

Example:
See the example for @CKDRV in Sample Program D, lines 38-39.

Test if Drive Assigned in DCT

Tests to determine whether a drive is defined in the Drive Code Table (DCT).

Entry Conditions:
A= 40 (X'28')
C=*logical drive number* (0-7)

Exit Conditions:
Success always.
If Z is set, the specified drive is already defined in the DCT.
If NZ is set, the specified drive is not defined in the DCT.

General:
Only AF is altered by this SVC.

Example:
See Sample Program D, lines 27-33.

Enter DEBUG

Forces the system to enter the DEBUG utility. Pressing (G) (ENTER) from the DEBUG monitor causes program execution to continue with the next instruction. If you want to use the functions in the extended debugger when DEBUG is entered in this fashion, you must issue the DEBUG (E) command (optionally with the @CMNDR SVC) before this SVC is executed.

Entry Conditions:
A=27 (X'1B')

General:
This SVC does not return unless (G) is entered in DEBUG.

Example:
See Sample Program A, lines 54-60.

Convert Decimal ASCII to Binary

Converts a decimal ASCII string to a 16-bit binary number. Overflow is not trapped. Conversion stops on the first out-of-range character.

Entry Conditions:
A = 96 (X'60')
HL = *pointer to decimal string*

Exit Conditions:
Success always.
BC = *binary conversion of ASCII string*
HL = *pointer to the terminating byte*
AF is altered by this SVC.

Example:
See Sample Program B, lines 88-95.

Directory Record Read

Reads a directory sector that contains the directory entry for a specified Directory Entry Code (DEC). The sector is placed in the system buffer and the register pair HL points to the first byte of the directory entry specified by the DEC.

Entry Conditions:
A = 87 (X'57')
B = *Directory Entry Code of the file*
C = *logical drive number* (0-7)

Exit Conditions:
Success, Z flag set.
HL = *pointer to directory entry specified by register B*
Failure, NZ flag set.
A = *error number*
HL is altered.

General:
AF is always altered.
If the drive does not contain a disk, this SVC may hang indefinitely waiting for formatted media to be placed in the drive. The programmer should perform a @CKDRV SVC before executing this call.
If the Directory Entry Code is invalid, the SVC may not return or it may return with the Z flag set and HL pointing to a random address. Care should be taken to avoid using the wrong value for the DEC in this call.

Example:
See Sample Program C, lines 152-174.

Directory Record Write

Writes the system buffer back to the disk directory sector that contains the directory entry of the specified DEC.

Entry Conditions:

A = 88 (X'58')
B = *Directory Entry Code of the file*
C = *logical drive number* (0-7)

Exit Conditions:

Success, Z flag set.
HL = *pointer to directory entry specified by register B*
Failure, NZ flag set.
A = *error number*
HL is altered.

General:

AF is always altered.

If the drive does not contain a disk, this SVC may hang indefinitely waiting for formatted media to be placed in the drive. The programmer should perform a @CKDRV SVC before executing this call.

If the Directory Entry Code is invalid, the SVC may not return or it may return with the Z flag set and HL pointing to a random address. Care should be taken to avoid using the wrong value for the DEC in this call.

Example:

See the example for @DIRRD in Sample Program C, lines 152-174.

8-Bit Divide

Performs an 8-bit unsigned integer divide.

Entry Conditions:
A = 93 (X'5D')
E = *dividend*
C = *divisor*

Exit Conditions:
Success always.
A = *quotient*
E = *remainder*
No other registers are altered.

Example:
See Sample Program B, lines 61-64.

16-Bit by 8-Bit Divide

Performs a division of a 16-bit unsigned integer by an 8-bit unsigned integer.

Entry Conditions:
A = 94 (X'5E')
HL = *dividend*
C = *divisor*

Exit Conditions:
Success always.
HL = *quotient*
A = *remainder*
No other registers are altered.

Example:
See Sample Program B, lines 105-109.

Do Directory Display / Buffer

Reads files from a disk directory or finds the free space on a disk. The directory information is either displayed on the screen (in five-across format) or sent to a buffer. The directory information buffer consists of 18 bytes per active, visible file: the first 16 bytes of the directory record, plus the ERN (ending record number). An X'FF' marks the buffer end.

Entry Conditions:

A = 34 (X'22')
C = *logical drive number* (0-7)
B selects one of the following functions:

If B = 0, the directory of the visible, non-system files on the disk in the specified drive is displayed on the screen. The filenames are displayed in columns, 5 filenames per line.

If B = 1, the directory is written to memory.
HL = *pointer to buffer to receive information*

If B = 2, a directory of the files on the specified drive is displayed for files that are visible, non-system, and match the extension partspec pointed to by HL.
HL = *partspec for the filename's extension*
This field must contain a valid 3-character extension, padded with dollar signs ($). For example, to display all visible, non-system files that have the letter 'C' as the first character of the extension, HL should point to the string "C$$".

If B = 3, a directory of the files on the specified drive is written to the buffer that is specified by HL for files that match the extension partspec pointed to by HL.
HL = *pointer to the 3-byte partspec and to the buffer to receive the directory records* (see general notes)
Keep in mind that the area pointed to by HL is shared. If you are using this buffer more than once, you have to re-create the partspec in the buffer before each call because the previous call will have erased the partspec by writing the directory records.

If B = 4, the disk name, original free space, and current free space on the disk is read.
HL = *pointer to a 20-byte buffer to receive information*

Exit Conditions:

Success, Z flag set.

If B = 1 or 3, the directory records have been stored.
HL = *pointer to the beginning of the buffer*

If B = 0 or 2, the filenames or matching filenames are displayed with 5 filenames per line.

If B = 4, the disk name and free space information are stored in the format:

Bytes 0-7 = Disk name. Disk name is padded on the right with blanks (X'20').
Bytes 8-15 = Creation date (the date the disk was formatted or was the target disk in a mirror image backup). The date is in the format MM/DD/YY.
Bytes 16-17 = Total K originally available in binary LSB-MSB format.
Bytes 18-19 = Free K available now in binary LSB-MSB format.

HL = *pointer to the beginning of the data area*

Failure, NZ flag set.

A = *error number*

General:

AF is the only register altered by this SVC.

The size of the buffer to receive directory records must be large enough to hold directory entries for the maximum number of files allowed on the drive and disk you specify. For example, if the drive is a hard disk, you must be able to store 256 directory entries, and each entry requires 18 bytes of storage. For more information on calculating the amount of space needed for this buffer, see the tables under "Directory Records." They give the maximum number of entries allowed on a given type of disk. You must add 2 records to this value when B = 1 to store the directory entry for DIR/SYS and BOOT/SYS.

Example:

See Sample Program E, lines 32-40.

Display Character

Outputs a byte to the video display. The byte is displayed at the current cursor position.

Entry Conditions:
A=2 (X'02')
C=*byte to display*

Exit Conditions:
Success, Z flag set.
A=*byte displayed*
Failure, NZ flag set.
A=*error number*

General:
DE is altered by this SVC.

Example:
See Sample Program C, lines 219-221.

Display Message Line

Displays a message line, starting at the current cursor position. The line must be terminated with either a carriage return (X'0D') or an ETX (X'03'). If an ETX terminates the line, the cursor is positioned immediately after the last character displayed.

Entry Conditions:
A = 10 (X'0A')
HL = *pointer to first byte of message*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
AF and DE are altered by this SVC.

Example:
See Sample Program C, lines 35-37.

Entry to Post an Error Message

Provides an entry to post an error message. If bit 7 of register C is set, the error message is displayed and return is made to the calling program. If bit 6 is not set, the extended error message is displayed. Under versions prior to 6.2 the error display is in the following format:

```
*** Errcod=xx, Error message string ***
     &lt;filespec or devspec&gt;
Referenced at X'dddd'
```

Under Version 6.2 the error display is in the following format:

```
**Error code = xx, Returns to X' dddd'
**Error message string
<filespec, devspec, or open FCB/DCB status>
Last SVC = nnn, Returned to X' rrrr'
```

dddd is the return address of the @ERROR SVC in the application program.
nnn is the last SVC executed before the @ERROR SVC request.
rrrr is the address the previous SVC returned to in the application program.

If bit 6 is set, then only the "Error message string" is displayed. This bit is ignored if bit 6 of SFLAG$ (the extended error message bit) is set. If bit 6 of CFLAG$ is set, then no error message is displayed. If bit 7 of CFLAG$ is set, then the "Error message string" is placed in a user buffer pointed to by register pair DE. See @FLAGS (SVC 101) for more information on SFLAG$ and CFLAG$.

Entry Conditions:
A = 26 (X'1A')
C = *error number with bits 6 and 7 optionally set*

Exit Conditions:
Success always.

General:
To avoid a looping condition that could result from the display device generating an error, do not check for errors after returning from @ERROR.
If you do not set bit 6 of register C, then you should execute this SVC only after an error has actually occurred.

Example:
See Sample Program C, lines 379-389.

Exit to TRSDOS

This is the normal program exit and return to TRSDOS. An error exit can be done by placing a non-zero value in HL. Values 1 to 62 indicate a primary error as described in TRSDOS Error Codes (Appendix A). (A non-zero value in HL causes an active JCL to abort.)

Entry Conditions:

A = 22 (X'16')
HL = *Return Code*
If HL = 0, then no error on exit.
If HL ≠ 0, then the @ABORT SVC returns X'FFFF' in HL automatically.

General:

This SVC does not return.

Example:

See Sample Program B, lines 206-207.

Set Up Default File Extension

Inserts a default file extension into the File Control Block if the file specification entered contains no extension. @FEXT must be done before the file is opened.

Entry Conditions:

A =79 (X'4F')
DE=*pointer to FCB*
HL=*pointer to default extension* (3 characters; alphabetic characters must be upper case and first character must be a letter)

Exit Conditions:

Success always.
AF and BC are altered by this SVC.
If the default extension is used, HL is also altered.

Example:

See Sample Program C, lines 111-132.

Point IY to System Flag Table

Points the IY register to the base of the system flag table. The status flags listed below can be referenced off IY. You can alter those bits marked with an asterisk (*). Bits without an asterisk are indicators of current conditions, or are unused or reserved.

Note: You may wish to save KFLAG$ and SFLAG$ if you intend to modify them in your program, and restore them on exit.

Entry Conditions:

A = 101 (X'65')

Exit Conditions:

Success always.

IY = *pointer to the following system information:*

IY − 1 Contains the overlay request number of the last system module resident in the system overlay region.

IY + 0 = AFLAG$ (allocation flag under Version 6.2 only)
Contains the starting cylinder number to be used when searching for free space on a diskette. It is normally 1. If the starting cylinder number is larger than the number of cylinders for a particular drive, 1 is used for that drive.

IY + 2 = CFLAG$

- * bit 7 — If set, then @ERROR will transfer the "Error message string" to your buffer instead of displaying it. The message is terminated with X'0D.'
- * bit 6 — If set, do not display system error messages 0-62. See @ERROR (SVC 26) for more information.
- * bit 5 — If set, sysgen is not allowed.
- * bit 4 — If set, then @CMNDR will execute only system library commands.
- bit 3 — If set, @RUN is requested from either the SET or SYSTEM (DRIVER =) commands.
- bit 2 — If set, @KEYIN is executing due to a request from SYS1.
- bit 1 — If set, @CMNDR is executing. This bit is reset by @EXIT and @CMNDI.
- * bit 0 — If set, HIGH$ cannot be changed using @HIGH$ (SVC 100). This bit is reset by @EXIT and @CMNDI.

IY + 3 = DFLAG$ (device flag)

- * bit 7 — "1" if GRAPHIC printer capability desired on screen print (CONTROL : causes screen print. See the SYSTEM (GRAPHIC) command under "Technical Information on TRSDOS Commands and Utilities.")
- bit 6 — "1" if KSM module is resident
- bit 5 — Currently unused
- bit 4 — "1" if MemDisk active
- bit 3 — Reserved
- bit 2 — "1" if Disk Verify is enabled
- * bit 1 — "1" if TYPE-AHEAD is active
- bit 0 — "1" if SPOOL is active

IY + 4 = EFLAG$ (ECI flag under Version 6.2 only)
Indicates the presence of an ECI program. If any of the bits are set, an ECI is used, rather than the SYS1 interpreter. The ECI program may use these bits as neccesary. However, at least one bit must be set or the ECI is not executed.

IY+5 = FEMSK$ (mask for port 0FEH)
IY+8 = IFLAG$ (international flag)
* bit 7 — If "1," 7-bit printer filter is active
If "0," normal 8-bit filters are present
* bit 6 — If "1," international character translation will be performed by printer driver
If "0," characters received by printer driver will be sent to the printer unchanged
bit 5 — Reserved for future languages
bit 4 — Reserved for future languages
bit 3 — Reserved for future languages
bit 2 — Reserved for future languages
bit 1 — If "1," German version of TRSDOS is present
bit 0 — If "1," French version of TRSDOS is present
If bits 5-0 are all zero, then USA version of TRSDOS is present.
IY+10 = KFLAG$ (keyboard flag)
bit 7 — "1" if a character is present in the type-ahead buffer
bit 6 — Currently unused
* bit 5 — "1" if CAPS lock is set
bit 4 — Currently unused
bit 3 — Currently unused
* bit 2 — "1" if (ENTER) has been pressed
* bit 1 — "1" if (SHIFT) (@) has been pressed (PAUSE)
* bit 0 — "1" if (BREAK) has been pressed
Note: To use bits 0-2, you must first reset them and then test to see if they become set.
IY+12 = MODOUT (image of port 0ECH)
IY+13 = NFLAG$ (network flag under Version 6.2)
bit 7 — Reserved for system use.
bit 6 — If set, the application program is in the task processor. Programmers must **not** modify this bit.
bit 5 — Reserved for system use.
bit 4 — Reserved for system use.
bit 3 — Reserved for system use.
bit 2 — Reserved for system use.
bit 1 — Reserved for system use.
* bit 0 — If set, the "file open bit" is written to the directory.
IY+14 = OPREG$ (memory management & video control image)
IY+17 = RFLAG$ (retry flag under Version 6.2 only)
Indicates the number of retrys for the floppy disk driver. This should be an even number larger than two.
IY+18 = SFLAG$ (system flag)
bit 7 — "1" if DEBUG is to be turned on
* bit 6 — "1" if extended error messages desired (see @ERROR for message format); overrides the setting of bit 6 of register C on @ERROR (SVC 26) and should be used only when testing
bit 5 — "1" if DO commands are being executed
* bit 4 — "1" if BREAK disabled
bit 3 — "1" if the hardware is running at 4 mhz (SYSTEM (FAST)). If "0," the hardware is running at 2 mhz (SYSTEM (SLOW)).
* bit 2 — "1" if LOAD called from RUN
* bit 1 — "1" if running an EXECute only file
* bit 0 — "1" specifies no check for matching LRL on file open and do not set file open bit in directory. This bit should be set just before executing an @OPEN (SVC 59) if you want to force the opened file to be READ only during current I/O operations. As soon as either call is executed, SFLAG$ bit 0 is reset. If you want to disable LRL checking on another file, you must set SFLAG$ bit 0 again.

IY + 19 = TFLAG$ (type flag under Version 6.2 only)
Identifies the Radio Shack hardware model. TFLAG$ allows programs to be aware of the hardware environment and the character sets available for the display. Current assignments are:

2 indicates Model II
4 indicates Model 4
5 indicates Model 4P
12 indicates Model 12

IY + 20 = UFLAG$ (user flag under Version 6.2 only)
May be set by application programs and is sysgened properly.

IY + 21 = VFLAG$
bit 7 — Reserved for system use
* bit 6 — "1" selects solid cursor, "0" selects blinking cursor
bit 5 — Reserved for system use
* bit 4 — "1" if real time clock is displayed on the screen
bits 0-3 — Reserved for system use

IY + 22 = WRINTMASK$ (mask for WRINTMASK port)
IY + 26 = SVCTABPTR$ (pointer to the high order byte of the SVC table address; low order byte = 00)
IY + 27 = Version ID byte (60H = TRSDOS version 6.0.x.x, 61H = TRSDOS version 6.1.x.x, etc.)
IY − 47 = Operating system release number. Provides a third and fourth character (12H = TRSDOS version x.x.1.2)
IY + 28
to
IY + 30 = @ICNFG vector
IY + 31
to
IY + 33 = @KITSK vector

@FNAME

Get Filename

Gets the filename and extension from the directory using the specified Directory Entry Code (DEC) for the file.

Entry Conditions:

A = 80 (X'50')
DE = *pointer to 15-byte buffer to receive filename/extension:drive, followed by a X'0D' as a terminator*
B = *DEC of desired file*
C = *logical drive number of drive containing file* (0-7)

Exit Conditions:

Success, Z flag set.
HL = *pointer to directory entry specified by register B*
Failure, NZ flag set.
A = *error number*
HL is altered.

General:

AF and BC are always altered.

If the drive does not contain a disk, this SVC may hang indefinitely waiting for formatted media to be placed in the drive. The programmer should perform a @CKDRV SVC before executing this call.

If the Directory Entry Code is invalid, the SVC may not return or it may return with the Z flag set and HL pointing to a random address. Care should be taken to avoid using the wrong value for the DEC in this call.

Example:

See Sample Program C, lines 274-286.

Assign File or Device Specification

Moves a file or device specification from an input buffer into a File Control Block (FCB). Conversion of lower case to upper case is made automatically.

Entry Conditions:
A =78 (X'4E')
HL=*pointer to buffer containing filespec or devspec*
DE=*pointer to 32-byte FCB or DCB*

Exit Conditions:
Success always.
If the Z flag is set, the file specification is valid.
HL=*pointer to terminating character*
DE=*pointer to start of FCB*
If the NZ flag is set, a syntax error was found in the filespec.
HL=*pointer to invalid character*
DE=*pointer to start of FCB*
A =*invalid character*

General:
AF and BC are altered.

Example:
See Sample Program C, lines 53-65.

Get One Byte From Device or File

Gets a byte from a logical device or a file. The DCB TYPE byte (DCB + 0, Bit 0) must permit a GET operation for this call to be successful.

Entry Conditions:
A = 3 (X'03')
DE = *pointer to DCB or FCB*

Exit Conditions:
Success, Z flag set.
A = *character read from the device or file*
Failure, NZ flag set. Test register A:
If A = 0, no character was available.
If A ≠ 0, A contains *error number.*

Example:
See the section "Device Driver and Filter Templates."

Get Device Control Block Address

Finds the location of a Device Control Block (DCB). If DE = 0 (no device name specified), HL returns the address of the first unused DCB found.

Entry Conditions:

A = 82 (X'52')
DE = *2-character device name* (E = first character, D = second character)

Exit Conditions:

Success, Z flag set. DCB was found.
HL = *pointer to start of DCB*
Failure, NZ flag set. No DCB was available.
A = *Error 8 (Device not available)*
HL is altered.

General:

AF is always altered by this SVC.

Example:

See the section "Device Driver and Filter Templates."

Get Drive Code Table Address

Gets the address of the Drive Code Table for the requested drive.

Entry Conditions:
A = 81 (X'51')
C = *logical drive number* (Ø-7)

Exit Conditions:
Success always.
IY = *pointer to the DCT entry for the specified drive*
AF is always altered by this SVC.

General:
If the drive number is out of range, the IY pointer will be invalid. This call does not return Z/NZ to indicate if the drive number specified is valid (Ø-7) or enabled.

Example:
See the example for @DCSTAT in Sample Program D, lines 27-33.

Get Memory Module Address

Locates a memory module, if the standard memory header is at the start of the module. The scanning starts with the system drivers in low memory, then moves to any high memory modules. If any routine is encountered that does not start with a proper header, scanning stops.

Entry Conditions:

A = 83 (X'53')
DE = *pointer to memory module name in upper case, terminated with any character in the range 00-31*

Exit Conditions:

Success always.
If the Z flag is set, the module was found.
HL = *pointer to first byte of memory header*
DE = *pointer to first byte after module name*
If the NZ flag is set, the module was not found.
HL is altered.

General:

AF is always altered by this SVC.

Example:

See Sample Program F, lines 144-154.

@HDFMT

Hard Disk Format

Passes a format drive command to a hard disk driver. If the hard disk controller accepts it as a valid command, then it formats the entire disk drive. If the hard disk controller does not accept it, then an error is returned. Radio Shack hardware does not currently support @HDFMT.

Entry Conditions:

A = 52 (X'34')
C = *logical drive number* (0-7)

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set.
A = *error number*

Convert Binary to Decimal ASCII

Converts a binary number in HL to decimal ASCII.

Entry Conditions:

A =97 (X'61')
HL = *number to convert*
DE= *pointer to 5-character buffer to hold converted number*

Exit Conditions:

Success always.
DE = *pointer to end of buffer + 1*
AF, BC, and HL are altered by this SVC.

Example:

See Sample Program B, lines 73-76.

Convert 1 Byte to Hex ASCII

Converts a 1-byte number to hexadecimal ASCII.

Entry Conditions:
A = 98 (X'62')
C = *number to convert*
HL = *pointer to a 2-character buffer to hold the converted number*

Exit Conditions:
Success always.
HL = *pointer to the end of buffer + 1*
Only AF is altered by this SVC.

Example:
See Sample Program B, lines 236-246.

Convert 2 Bytes to Hex ASCII

Converts a 2-byte number to hexadecimal ASCII.

Entry Conditions:
A =99 (X'63')
DE= *number to convert*
HL= *pointer to 4-character buffer to hold converted number*

Exit Conditions:
Success always.
HL= *pointer to end of buffer + 1*
Only AF is altered by this SVC.

Example:
See Sample Program B, lines 248-258.

Get or Alter HIGH$ or LOW$

Provides the means to read or alter the HIGH$ and LOW$ values.

Note: HIGH$ must be greater than LOW$. LOW$ is reset to X'2FFF' by @EXIT, @ABORT, and @CMNDI.

Entry Conditions:
A = 100 (X'64')
B selects HIGH$ or LOW$
If B = 0, SVC deals with HIGH$
If B ≠ 0, SVC deals with LOW$
HL selects one of the following functions:
If HL = 0, the current HIGH$ or LOW$ is returned
If HL ≠ 0, then HIGH$ or LOW$ is set to the value in HL

Exit Conditions:
Success, Z flag set.
HL = *current HIGH$ or LOW$*. If HL ≠ 0 on entry, then HIGH$ or LOW$ is now set to that value.
Failure, NZ flag set.
A = *error number*

General:
If bit 0 of CFLAG$ is set (see @FLAGS), then HIGH$ cannot be changed with this call. The call returns error 43, "SVC parameter error."

Example:
See Sample Program F, lines 75-86.

Open or Initialize File

Opens a file. If the file is not found, this SVC creates it according to the file specification.

Entry Conditions:
A =58 (X'3A')
HL = *pointer to 256-byte disk I/O buffer*
DE= *pointer to FCB containing the file specification*
B = *Logical Record Length to be used while file is open*

Exit Conditions:
Success, Z flag set. File was opened or created.
The CF flag is set if a new file was created.
Failure, NZ flag set.
A = *error number*

General:
Only AF is altered by this SVC.
The file open bit is set in the directory if the access level is UPDATE or greater.

Example:
See Sample Program C, lines 260-272.

Reboot the System

Does a software reset. Floppy drive 0 must contain a system disk. @IPL uses the standard boot sequence, the same as for a hard reset (pressing the reset button). Memory locations X'41E5'-X'4225' and X'4300'-X'43FF' are altered during the boot of the machine.

Entry Conditions:
A = 0 (X'00')

General:
This SVC does not return.

Scan Keyboard and Return

Scans the keyboard and returns a character if a key is pressed. If no key is pressed, a zero value is returned.

Entry Conditions:
A = 8 (X'08')

Exit Conditions:
Success, Z flag set.
A = *character pressed*
Failure, NZ set.
If A = 0, no character was available.
If A ≠ 0, then A contains *error number.*

General:
DE is altered by this SVC.

Example:
See Sample Program C, lines 198-200.

Scan *KI Device, Wait for Character

Scans the *KI device and returns with a character. It does not return until a character is input to the device.

Note: The system suspends execution of the program that issued the SVC until a character can be obtained. Background tasks will continue to run normally.

Entry Conditions:
A=1 (X'01')

Exit Conditions:
Success, Z flag set.
A=*character entered*
Failure, NZ flag set.
A=*error number*

General:
DE is altered by this SVC.

Example:
See Sample Program B, lines 202-203.

Accept a Line of Input

Accepts a line of input until terminated by either an (ENTER) or a (BREAK). Entries are displayed on the screen, starting at the current cursor position. Backspace, tab, and line delete are supported. If JCL is active, the line is fetched from the active JCL file.

Entry Conditions:
A =9 (X'09')
HL = *pointer to user line buffer of length B + 1*
B = *maximum number of characters to input*
C = 0

Exit Conditions:
Success, Z flag set.
HL = *pointer to start of buffer*
B = *actual number of characters input*
CF is set if (BREAK) terminated the input.
Failure, NZ flag set.
A = *error number*

General:
DE and C are altered by this SVC.

Example:
See Sample Program C, lines 39-47.

Remove Currently Executing Task

When called by an executing task driver, removes the task assignment from the task table and returns to the foreground application that was interrupted.

Entry Conditions:
A = 32 (X'20')

General:
This SVC does not return.

Example:
See the example for @RMTSK in Sample Program F, lines 134-142.

Load Program File

Loads a program file. The file must be in load module format.

Entry Conditions:
A = 76 (X'4C')
DE = *pointer to FCB containing filespec of the file to load*

Exit Conditions:
Success, Z flag set.
HL = *transfer address retrieved from file*
Failure, NZ flag set.
A = *error number*

Example:
See Sample Program A, lines 50-56.

Calculate Current Logical Record Number

Returns the current logical record number.

Entry Conditions:
A = 63 (X'3F')
DE = pointer to the file's FCB

Exit Conditions:
Success, Z flag set.
BC = *logical record number*
Failure, NZ flag set.
A = *error number*

General:
AF is altered by this SVC.

Example:
See Sample Program C, lines 305-311.

Calculate the EOF Logical Record Number

Returns the EOF (End of File) logical record number.

Entry Conditions:
A = 64 (X'40')
DE = *pointer to FCB for the file to check*

Exit Conditions:
Success, Z flag set.
BC = *the EOF logical record number*
Failure, NZ flag set.
A = *error number*

General:
Only AF is altered by this SVC.

Example:
See the example for @LOC in Sample Program C, lines 305-311.

Issue Log Message

Issues a log message to the Job Log. The message can be any character string terminating with a carriage return (X'0D').

Entry Conditions:
A = 11 (X'0B')
HL = *pointer to first character in message line*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
Only AF is altered by this SVC.

Example:

```
        LD      HL,TEXT     ;Point at message to output
        LD      A,@LOGER    ;and output it to the Job
                            ;Log
        RST     28H         ;Call the @LOGER SVC
        ...
TEXT:   DEFM    'This is a message for the Job Log'
        DEFB    0DH         ;Message must be terminated
                            ;with an <ENTER>.
```

Display and Log Message

Displays and logs a message. Performs the same function as @DSPLY followed by @LOGER.

Entry Conditions:
A = 12 (X'0C')
HL = *pointer to first character in message line*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
Only AF is altered by this SVC.
To avoid a looping condition that could result from the display device generating an error, no error checking should be done after returning from @LOGOT.

Example:

```
        LD    HL,TEXT   ;Point at message to output
        LD    A,@LOGOT  ;and output it to the Job
                        ;Log AND the display
        RST   28H       ;Call the @LOGOT SVC
        ...
TEXT:   DEFM  'This message will be displayed both in'
        DEFM  'the Job Log and on the display.'
        DEFB  0DH       ;Must terminate text with an
                        ;<ENTER>.
```

Send Message to Device

Sends a message line to any device or file.

Entry Conditions:
A = 13 (X'0D')
DE = *pointer to DCB or FCB of device or file to receive output*
HL = *pointer to message line terminated with X'0D' or X'03'*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
Only AF is altered by this SVC.

Example:

```
        LD     HL,TEXT    ;Point at message to output
        LD     DE,DCBP    ;Point at the device control
                          ;block for our device
        LD     A,@MSG     ;and write this text to it
        RST    28H        ;Call the @MSG SVC
        ...
TEXT:   DEFM   'D555-555<LOGIN USER>' ;Text to write to
                          ;this device. In this case,
                          ;it is a dialing modem.
        DEFB   03H        ;Terminate the message
```

8-Bit Multiplication

Performs an 8-bit by 8-bit unsigned integer multiplication. The resultant product must fit into an 8-bit field.

Entry Conditions:
A = 90 (X'5A')
C = *multiplicand*
E = *multiplier*

Exit Conditions:
Success always.
A = *product*
DE is altered by this SVC.

Example:
See Sample Program B, lines 150-153.

16-Bit by 8-Bit Multiplication

Performs an unsigned integer multiplication of a 16-bit multiplicand by an 8-bit multiplier. The resultant product is stored in a 3-byte register field.

Entry Conditions:

A = 91 (X'5B')
HL = *multiplicand*
C = *multiplier*

Exit Conditions:

Success always.

HL = *two high-order bytes of product*
A = *low-order byte of product*
DE is altered by this SVC.

Example:

See Sample Program B, lines 183-187.

Open Existing File or Device

Opens an existing file or device.

Entry Conditions:

A =59 (X'3B')
HL=*pointer to 256-byte disk I/O buffer*
DE=*pointer to FCB or DCB containing filespec or devspec*
B =*logical record length for open file*

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set.
A=*error number*

General:

AF is altered by this SVC.
The file open bit is set in the directory if the access level is UPDATE or greater.

Example:

See Sample Program C, lines 134-150.

Parse Parameter String

Parses an optional parameter string. Its primary function is to parse command parameters contained in a command line starting with a parenthesis. The acceptable parameter format is:

PARM = X'nnnn'....hexadecimal entry
PARM = nnnnn ....decimal entry
PARM = "string" ...alphanumeric entry
PARM = flag ....ON, OFF, Y, N, YES, or NO

Note: Entering a parameter with no equal sign or value is the same as using PARM = ON. Entering PARM = with no value is the same as using PARM = OFF.

Entry Conditions:

A = 17 (X'11')
DE = *pointer to beginning of your parameter table*
HL = *pointer to command line to parse* (the parameter string is enclosed within parentheses)

Exit Conditions:

Success always.
If Z is set, either valid parameters or no parameters were found.
If NZ is set, a bad parameter was found.

General:

NZ is not returned if parameter types other than those specified are entered. The application must check the validity of the response byte.

The valid parameters are contained in a user table which must be in one of the following formats. (Parameter names must consist of alphanumeric characters, the first of which is a letter.)

For use with TRSDOS Version 6, use this format:

The parameter table starts with a single byte X'80'. Each parameter is stored in a variable length field as described below.

1) Type Byte (Type and length byte)
Bit 7 — If set, accept numeric value
Bit 6 — If set, accept flag parameter
Bit 5 — If set, accept "string" value
Bit 4 — If set, accept first character of name as abbreviation
Bits 3-0 — Length of parameter name

2) Actual Parameter Name

3) Response byte (Type and length found)
Bit 7 — Numeric value found
Bit 6 — Flag parameter found
Bit 5 — String parameter found
Bits 4-0 — Length of parameter entered. If length is 0 and the 2-byte vector points to a quotation mark (X'22'), then the parameter was a null string. Otherwise, a length of 0 indicates that the parameter was longer than 31 characters.

4) 2-byte address vector to receive the parsed parameter values.

The 2-byte memory area pointed to by the address field of your table receives the value of PARM if PARM is non-string. If a string is entered, the 2-byte memory area receives the address of the first byte of "string." The entries ON, YES, and Y return a value of X'FFFF'; OFF, NO, and N return X'0000'. If a parameter name is specified on the command line and is followed by an equal sign and no value, then X'0000' or NO is returned. If a parameter name is used on the command line without the equal sign, then a value of X'FFFF' or ON is assumed. For any allowed parameter that is completely omitted on the command line, the 2-byte area remains unchanged and the response byte is 0.

The parameter table is terminated with a single byte X'00'.

For compatibility with LDOS 5.1.3, use this format:

A 6-character "word" left justified and padded with blanks followed by a 2-byte address to receive the parsed values. Repeat word and address for as many parameters as are necessary. You must place a byte of X'00' at the end of the table.

Example:

```
          LD    HL,COMAND ;Point at command buffer
          LD    DE,PARM   ;Point at parameter list
          LD    A,@PARAM  ;Parse the items on the
                          ;command line
          RST   28H       ;Call the @PARAM SVC
          JR    NZ,ERROR  ;An error occurred (not
                          ;included here)
          LD    A,(RESP)  ;Get response code
          AND   040H      ;Test response flags
          JR    Z,BAD     ;User specified something
                          ;like UPDATE=X'1234' or
                          ;UPDATE="HELLO"
          LD    A,(VAL)   ;Get 1st byte of VAL word
          OR    A         ;Test the value
          JR    Z,OFF     ;UPDATE=OFF or UPDATE=NO was
                          ;specified
          JR    ON        ;UPDATE=ON or UPDATE=YES was
                          ;specified
          ...
COMAND:   DEFS  80        ;Area where command is
                          ;stored
PARM:     DEFB  80H       ;Table header code
          DEFB  40H+6     ;40 says we want a flag
                          ;(YES/NO). 6 is length of
                          ;the parameter name
          DEFM  'UPDATE'  ;Parameter name
RESP:     DEFB  0         ;Response area
          DEFW  VAL       ;Vector to VAL
          DEFB  0         ;End of Table code
VAL:      DEFS  2         ;Area to receive a parameter
                          ;value
```

Suspend Program Execution

Suspends program execution for a specified period of time and goes into a "holding" state. The delay is at least 14.3 microseconds per count.

Entry Conditions:
A = 16 (X'10')
BC = *delay count*

Exit Conditions:
Success always.

Example:

```
LD    BC,36A2H  ;Wait for about 200 milli-
                ;seconds. 14.3 usecs *
                ;13986 is approx. 200
                ;msecs
LD    A,@PAUSE  ;Suspend execution
RST   28H       ;Call the @PAUSE SVC
```

Position to End Of File

Positions an open file to the End Record Number (ERN). An end-of-file-encountered error (X'1C') is returned if the operation is successful. Your program may ignore this error.

Entry Conditions:

A = 65 (X'41')
DE = *pointer to FCB of the file to position*

Exit Conditions:

NZ flag always set.
If A = X'1C,' then success.
If A ≠ X'1C,' then failure.
A = *error number*

General:

AF is always altered by this SVC.

Example:

See the example for @LOC in Sample Program C, lines 305-311.

Position File

Positions a file to a logical record. This is useful for positioning to records of a random access file.

When the @POSN routine is used, Bit 6 of FCB + 1 is automatically set. This ensures that the EOF (End Of File) is updated when the file is closed only if the NRN (Next Record Number) exceeds the current ERN (End Record Number).

Note that @POSN must be used for *each* write, even if two records are side by side.

Entry Conditions:
A = 66 (X'42')
DE = *pointer to FCB for the file to position*
BC = *the logical record number*

Exit Conditions:
If Z flag is set or A = X'1C' or X'1D', then success.
The file was positioned.
Otherwise, failure.
A = *error number*

General:
AF is always altered by this SVC.

Example:
See the example for @LOC in Sample Program C, lines 305-311.

Prints Message Line

Outputs a message line to the printer. The line must be terminated with either a carriage return (X'0D') or an ETX (X'03').

Entry Conditions:
A = 14 (X'0E')
HL = *pointer to message to be output*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
AF and DE are altered by this SVC.

Example:

```
        LD     HL,TEXT    ;Text to be output to the
                          ;printer
        LD     A,@PRINT   ;Write this message to the
                          ;printer device
        RST    28H        ;Call the @PRINT SVC
        ...
TEXT:   DEFB   0CH        ;Do a Top of Form
        DEFM   'Report continued                Page
        DEFB   3          ;Terminate with a <ETX> or
                          ;an <ENTER>
```

Send Character to Printer

Outputs a byte to the line printer.

Entry Conditions:
A = 6 (X'Ø6')
C = *character to print*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
AF and DE are altered by this SVC.
If the line printer is attached but becomes unavailable (out of paper, out of ribbon, turned off, off-line, buffer full, etc.), the printer driver waits approximately ten seconds. If the printer is still not ready, a "Device not available" error is returned.

Example:

```
        LD      A,(PAGE)    ;Get the page number
        ADD     A,'Ø'       ;Make it ASCII
        LD      C,A         ;Put the value here
        LD      A,@PRT      ;Write this character to the
                            ;printer
        RST     28H         ;Call the @PRT SVC
        ...
PAGE:   DEFB    2           ;Start with page 2
```

Write One Byte to Device or File

Outputs a byte to a logical device or file. The DCB TYPE byte (DCB + 0, Bit 1) must permit PUT operation.

Entry Conditions:

A = 4 (X'04')
DE = *pointer to DCB or FCB of the output device*
C = *byte to output*

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:

AF is always altered by this SVC.

Example:

See the section "Device Driver and Filter Templates."

Get Directory Record or Free Space

Reads the directory information of visible files from a disk directory, or gets the amount of free space on a disk.

Entry Conditions:

A =35 (X'23')
HL = *pointer to RAM buffer to receive information*
B = *logical drive number* (0-7)
C selects one of the following functions:
If C = 0, get directory records of all visible files.
If C = 255, get free space information.
If C = 1-254, get a single directory record (see below).

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set.
A = *error number*

Each directory record requires 22 bytes of space in the buffer. If C = 0, one additional byte is needed to mark the end of the buffer.

For single directory records, the number in the C register should be one less than the desired directory record. For example, if C = 1, directory record 2 is fetched and put in the buffer. If a single record request is for an inactive record or an invisible file, the A register returns an error code 25 (File access denied).

The directory information is placed in the buffer as follows:

Byte	Contents
00-14	filename/ext:d (left justified, padded with spaces)
15	protection level, 0 to 6
16	EOF offset byte
17	logical record length, 0 to 255
18-19	ERN of file
20-21	file size in K (1024-byte blocks)
22	LAST RECORD ONLY. Contains " + " to mark buffer end.

If C = 255, HL should point to a 4-byte buffer. Upon return, the buffer contains:

Bytes 00-01 Space in use in K, stored LSB, MSB
Bytes 02-03 Space available in K, stored LSB, MSB

Example:

See the example for @DODIR in Sample Program E, lines 32-40.

@RDHDR

Read a Sector Header

Reads the next ID header when supported by the controller driver. The floppy disk driver supplied treats this as a @RDSEC (SVC 49).

Entry Conditions:
A = 48 (X'30')
HL = *pointer to buffer to receive the data*
D = *cylinder to read*
C = *logical drive number*
E = *sector to read*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

Example:
See the example for @RDSEC in Sample Program D, lines 63-66.

Read Sector

Transfers a sector of data from the disk to your buffer.

Entry Conditions:

A = 49 (X'31')
HL = *pointer to the buffer to receive the sector*
D = *cylinder to read*
E = *sector to read*
C = *logical drive number* (0-7)

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:

Only AF is altered by this SVC

Example:

See Sample Program D, lines 63-66.

Read System Sector

Reads the specified system (directory) sector. If the cylinder number in register D is not the directory cylinder, the value in D is changed to reflect the real directory cylinder and the sector is then read.

Entry Conditions:
A = 85 (X'55')
HL = *pointer to the buffer to receive the sector*
D = *cylinder to read*
E = *sector to read*
C = *logical drive number* (0-7)

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
Only AF is altered by this SVC.

Example:
See Sample Program D, lines 78-92.

Read a Track

Reads an entire track when supported by the controller driver. The floppy disk driver supplied treats this as a @RDSEC (SVC 49) and does not do a track read.

Entry Conditions:

A =51 (X'33')
HL = *pointer to buffer to receive the sector*
D = *track to read*
C = *logical drive number*
E = *sector to read*

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:

AF is altered by the supplied floppy disk driver.

Example:

See the example for @RDSEC in Sample Program D, lines 63-66.

Read a Record

Reads a logical record from a file. If the LRL defined at open time was 256 (specified by 0), then the NRN sector is transferred to the buffer established at open time. For LRL between 1 and 255, the next logical record is placed into a user record buffer, UREC. The 3-byte NRN is updated after the read operation.

Entry Conditions:
A =67 (X'43')
DE=*pointer to FCB for the file to read*
HL=*pointer to user record buffer UREC* (needed if LRL=1-255; unused if LRL=256)

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A=*error number*

Example:
See Sample Program C, lines 300-304.

@REMOV

Remove File or Device

Removes a file or device.

If a file is to be removed, the File Control Block must be in an open condition. When this SVC is performed, the file's directory is updated and the space occupied by the file is deallocated.

If a device was specified, the device is closed. To remove a device, use the REMOVE library command.

Entry Conditions:
A = 57 (X'39')
DE = *pointer to FCB or DCB to remove*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

Example:
See Sample Program C, lines 223-231.

@RENAM

Rename File or Device

Changes a file's filename and/or extension.

Entry Conditions:

A =56 (X'38')

DE=*pointer to an FCB containing the file's current name*
This FCB must be in a closed state.

HL=*pointer to new filename string terminated with a X'0D' or X'03'.* This filespec must be in upper case and must be a valid filespec. You can convert the filespec to upper case and check its validity by using the @FSPEC SVC before using @RENAM.

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set.
A=*error number*

General:

After the call is completed, the FCB pointed to by DE is altered.
Only AF is altered by this SVC.

Example:

```
      LD    DE,FCB          ;Point at a closed FCB
                            ;containing the old
                            ;filespec
      LD    HL,NEW          ;Point to the new filespec
                            ;to use
      LD    A,@RENAM        ;Change the name of the
                            ;file
      RST   28H             ;Call the @RENAM SVC
      ...
FCB:  DEFS  32              ;A File Control Block used
                            ;by the @RENAM SVC. In
                            ;this example, it is
                            ;assumed that an @FSPEC
                            ;SVC has loaded a filespec
                            ;into the FCB before the
                            ;@RENAM SVC is performed.
NEW:  DEFM  'NEWNAME/TXT'   ;The new filespec for the
                            ;file
      DEFB  0DH             ;Terminate the filespec
```

Rewind File to Beginning

Rewinds a file to its beginning and resets the 3-byte NRN to 0. The next record to be read or written sequentially is the first record of the file.

Entry Conditions:
A =68 (X'44')
DE = *pointer to FCB for the file to rewind*

Exit Conditions:
Success, Z flag set. File positioned to record number 0.
Failure, NZ flag set.
A = *error number*

General:
AF is always altered by this SVC.

Example:
See the example for @LOC in Sample Program C, lines 305-311.

Remove Interrupt Level Task

Removes an interrupt level task from the Task Control Block table.

Entry Conditions:
A=30 (X'1E')
C=*task slot assignment to remove* (0-11)

Exit Conditions:
Success always.
HL and DE are altered by this SVC.

Example:
See Sample Program F, lines 134-142.

@RPTSK

SVC Number 31

Replace Task Vector

Exits the task process executing and replaces the currently executing task's vector address in the Task Control Block table with the address following the SVC instruction. Return is made to the foreground application that was interrupted.

Entry Conditions:
A=31 (X'1F')

General:
This SVC does not return.

Example:

```
        LD      A,RPTSK ;Replace this task with the
                        ;one located at the
                        ;following address:
        RST     28H     ;Call the @RPTSK SVC
NEWADD: DEFW    0       ;Address of the new task is
                        ;loaded here. This word
                        ;must be immediately after
                        ;the @RPTSK SVC. The label
                        ;NEWADD is present only to
                        ;allow the address to be
                        ;stored.
```

Reread Sector

Forces a reread of the current sector to occur before the next I/O request is performed. Its most probable use is in applications that reuse the disk I/O buffer for multiple files, to make sure that the buffer contains the proper file sector. This routine is valid only for byte I/O or blocked files. Do not use it when positioned at the start of a file.

Entry Conditions:
A = 69 (X'45')
DE = *pointer to FCB for the file to reread*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
AF is always altered by this SVC.

Example:

```
        LD      DE,FCB          ;Point to File Control Block
                                ;of the file that requires
                                ;the re-read
        LD      A,@RREAD        ;Before next I/O, reload
                                ;the current sector into
                                ;the system buffer for
                                ;this file
        RST     28H             ;Call the @RREAD SVC
```

Test for Drive Busy

Performs a test of the last selected drive to see if it is in a busy state. If busy, it is re-selected until it is no longer busy.

Entry Conditions:
A = 47 (X'2F')
C = *logical drive number* (0-7)

Exit Conditions:
Success always.
Only AF is altered by this SVC.

Example:

```
        LD      C,1         ;Test Drive 1 to see if it
                            ;is busy.
        LD      A,@RSLCT    ;If it is, continue
                            ;selecting it
        RST     28H         ;Call the @RSLCT SVC
```

Issue FDC RESTORE Command

Issues a disk controller RESTORE command.

Entry Conditions:
A = 44 (X'2C')
C = *logical drive number* (0-7)

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

Example:
See the example for @CKDRV in Sample Program D, lines 38-39.

Run Program

Loads and executes a program file. If an error occurs during the load, the system prints the appropriate message and returns.

Entry Conditions:

A = 77 (X'4D')
DE = *pointer to FCB containing the filespec of the file to RUN*
Note: The FCB must be located where the program being loaded will not overwrite it.

Exit Conditions:

Success, the new program is loaded and executed.
Failure, the error is displayed and return is made to your program.
HL = *return code* (See the section "Converting to TRSDOS Version 6" for information on return codes.)

General:

HL is returned unchanged if no error occurred and can be used as a pointer to a command line.

Example:

See Sample Program A, lines 62-74.

@RWRIT

Rewrite Sector

Rewrites the current sector, following a write operation. The @WRITE function advances the NRN after the sector is written. @RWRIT decrements the NRN and writes the disk buffer again. Do not use @RWRIT when positioned to the start of a file.

Entry Conditions:
A = 70 (X'46')
DE = *pointer to FCB for the file to rewrite*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

Example:

```
LD    DE,FCB     ;Point to the File Control
                 ;Block
LD    A,@RWRIT   ;Perform a re-write of the
                 ;current sector
RST   28H        ;Call the @RWRIT SVC
```

Seek a Cylinder

Seeks a specified cylinder and sector. @SEEK does not return an error if you specified a non-existent drive or an invalid cylinder. @SEEK performs no action if the specified drive is a hard disk.

Note: Seek of a sector is not supported by TRS-80 hardware. An implied seek is included in sector reads and writes.

Entry Conditions:
A = 46 (X'2E')
C = *logical drive number*
D = *cylinder to seek*
E = *sector to seek*

Exit Conditions:
Success always.
Only AF is altered by this SVC.

@SEEKSC

Seek Cylinder and Sector

Seeks the cylinder and sector corresponding to the next record of the specified file. (This is done by examining the NRN field of the FCB.) No error is returned on physical seek errors.

Entry Conditions:
A =71 (X'47')
DE=*pointer to the file's FCB*

Exit Conditions:
Success always.

Example:

```
        LD      DE,FCB      ;Point to the File Control
                            ;Block
        LD      A,@SEEKSC   ;Cause the next sector to be
                            ;SEEKed before it is
                            ;actually needed
        RST     28H         ;Call the @SEEKSC SVC
```

Skip a Record

Causes a skip past the next logical record. Only the record number contained in the FCB is changed; no physical I/O takes place.

Entry Conditions:
A =72 (X'48')
DE = *pointer to FCB for the file to skip*

Exit Conditions:
If the Z flag is set or if A = X'1C' or X'1D', then the operation was successful. Otherwise, A = *error number*. If A = X'1C' is returned, the file pointer is positioned at the end of the file. Any Appending operations would be performed here. If A = X'1D' is returned, the file pointer is positioned beyond the end of the file.

General:
AF is altered by this SVC.
BC contains the current record number. This is the same value as that returned by the @LOC SVC.

Example:
See the example for @LOC in Sample Program C, lines 305-311.

Select a New Drive

Selects a drive. The time delay specified in your configuration (SYSTEM (DELAY = Y/N)) is made if the drive selection requires it.

Entry Conditions:

A = 41 (X'29')
C = *logical drive number* (0-7)

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:

Only AF is altered by this SVC.

Sound Generation

Generates sound using specified tone and duration codes. Interrupts are disabled during execution.

Entry Conditions:
A = 104 (X'68')
B = *function code*
bits 0-2: tone selection (0-7 with 0 = highest and 7 = lowest)
bits 3-7: tone duration (0-31 with 0 = shortest and 31 = longest)

Exit Conditions:
Success always.
Only AF is altered by this SVC.

Example:
See Sample Program B, lines 43-45.

Issue FDC STEP IN Command

Issues a disk controller STEP IN command. This moves the drive head to the next higher-numbered cylinder. @STEPI is intended for sequential read/write operations, such as disk formatting.

Entry Conditions:
A = 45 (X'2D')
C = *logical drive number*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
Only AF is altered by this SVC.

@TIME

SVC Number 19

Get Time

Gets the system time in display format (HH:MM:SS).

Entry Conditions:

A = 19 (X'13')
HL = *pointer to buffer to receive the time string*

Exit Conditions:

Success always.

HL = *pointer to the end of buffer + 1*
DE = *pointer to start of TIME$ storage area in TRSDOS*
AF and BC are altered by this SVC.

Example:

See the example for @DATE in Sample Program F, lines 252-253.

Video Functions

Performs various functions related to the video display. The B register is used to pass the function number.

Entry Conditions:

A = 15 (X'0F')

B selects one of the following functions:

If B = 1, return the character at the screen position specified by HL.
H = *row on the screen* (0-23), where 0 is the top row
L = *column on the screen* (0-79), where 0 is the leftmost column

If B = 2, display the specified character at the position specified by HL.
C = *character to be displayed*
H = *row on the screen* (0-23), where 0 is the top row
L = *column on the screen* (0-79), where 0 is the leftmost column

If B = 3, move the cursor to the position specified by HL. This is done even if the cursor is not currently displayed.
H = *row on the screen* (0-23), where 0 is the top row
L = *column on the screen* (0-79), where 0 is the leftmost column

If B = 4, return the current position of the cursor.

If B = 5, move a 1920-byte block of data to video memory.
HL = *pointer to 1920-byte buffer to move to video memory*

If B = 6, move a 1920-byte block of data from video memory to a buffer you supply. In 40 line by 24 character mode, there must be a character in each alternating byte for proper display.
HL = *pointer to 1920-byte buffer to store copy of video memory* HL must be in the range X'23FF' < HL < X'EC01.

If B = 7, scroll protect the specified number of lines from the top of the screen.
C = *number of lines to scroll protect* (0-7). Once set, scroll protect can be removed only by executing @VDCTL with B = 7 and C = 0, or by resetting the system. Clearing the screen with (SHIFT)(CLEAR) erases the data in the scroll protect area, but the scroll protect still exists.

If B = 8, change cursor character to specified character. If the cursor is currently not displayed, the character is accepted anyway and is used as the cursor character when it is turned back on. The default cursor character is an underscore (X'5F') under Version 6.2 and a X'B0' under previous versions.
C = *character to use as the cursor character*

If B = 9, (under Version 6.2 only) transfer 80 characters to or from the screen.
If C = 0, move characters from the buffer to the screen
If C = 1, move characters from the screen to the buffer
H = *row on the screen*
DE = *pointer to 80 byte buffer*

Note: The video RAM area in the Models 4 and 4P is 2048 bytes (2K). The first 1920 bytes can be displayed. The remaining bytes contain the type-ahead buffer and other system buffers.

Exit Conditions:

If B = 1:

Success, Z flag set.

A = *character found at the location specified by HL*

DE is altered.

Failure, NZ flag set.

A = *error number*

If B = 2:

Success, Z flag set.

DE is altered.

Failure, NZ flag set.

A = *error number*

If B = 3:

Success, Z flag set.

DE and HL are altered.

Failure, NZ flag set.

A = *error number*

If B = 4:

Success always.

HL = *row and column position of the cursor.* H = row on the screen (0-23), where 0 is the top row; L = column on the screen (0-79), where 0 is the leftmost column.

If B = 5:

Success always.

HL = *pointer to the last byte moved to the video + 1*

BC and DE are altered.

If B = 6:

Success always.

BC, DE, and HL are altered.

If B = 7:

Success always.

BC and DE are altered.

If B = 8:

Success always.

A = *previous cursor character*

DE is altered.

If B = 9 (under Version 6.2 only):

Success, Z flag set.

BC, HL, DE are altered.

Failure, NZ flag set because H is out of range.

A = error code 43 (X'2B').

General:

Functions 5, 6, and 7 do not do range checking on the entry parameters.

If HL is not in the valid range in functions 5 and 6, the results may be unpredictable.

Only function 3 (B = 3) moves the cursor.

If C is greater than 7 in function 7, it is treated as modulo 8.

AF and B are altered by this SVC.

Example:

See Sample Program F, lines 304-327.

Write and Verify a Record

Performs a @WRITE operation followed by a test read of the sector (if the write required physical I/O) to verify that it is readable.

If the logical record length is less than 256, then the logical record in the user buffer UREC is transferred to the file. If the LRL is equal to 256, a full sector write is made using the disk I/O buffer identified at file open time.

Entry Conditions:

A = 73 (X'49')
DE = *pointer to FCB for the file to verify*

Exit Conditions:

Success, Z flag set.
HL = *pointer to user buffer containing the logical record*
Failure, NZ flag set.
A = *error number*

General:

Only AF is altered by this SVC.

Example:

See Sample Program C, lines 338-346.

Verify Sector

Verifies a sector without transferring any data from disk.

Entry Conditions:

A = 50 (X'32')
D = *cylinder to verify*
E = *sector to verify*
C = *logical drive number* (0-7)

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set
A = *error number*

General:

AF is always altered by this SVC.

If the sector is a system sector, the sector is readable if an error 6 is returned; any other error number signifies an error has occurred.

Example:

See the example for @WRSEC in Sample Program D, lines 68-76.

@WEOF

Write End Of File

Forces the system to update the directory entry with the current end-of-file information.

Entry Conditions:

A = 74 (X'4A')
DE = *pointer to the FCB for the file to WEOF*

Exit Conditions:

Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:

AF is always altered by this SVC.

Example:

```
LD     DE,FCB     ;Point at the File Control
                  ;Block
LD     A,@WEOF    ;Force the directory entry
                  ;to be updated now,
                  ;instead of when the file
                  ;is closed
RST    28H        ;Call the @WEOF SVC
```

Locate Origin of SVC

Used to resolve the relocation address of the calling routine.

Entry Conditions:
A = 7 (X'07')

Exit Conditions:
Success always.
HL = *pointer to address following RST 28H instruction*
AF is always altered by this SVC.

Example:
See Sample Program F, lines 36-60.

Write a Record

Causes a write to the next record identified in the File Control Block.

If the logical record length is less than 256, then the logical record in the user buffer UREC is transferred to the file. If the LRL is equal to 256, a full sector write is made using the disk I/O buffer identified at file open time.

Entry Conditions:
A =75 (X'4B')
HL=*pointer to user record buffer UREC* (unused if LRL=256)
DE=*pointer to FCB for the file to write*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A=*error number*

General:
AF is always altered by this SVC.

Example:
See the example for @VER in Sample Program C, lines 338-346.

Write a Sector

Writes a sector to the disk.

Entry Conditions:
A =53 (X'35')
HL= *pointer to the buffer containing the sector of data*
D = *cylinder to write*
E = *sector to write*
C = *logical drive number* (0-7)

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A= *error number*

General:
Only AF is altered by this SVC.

Example:
See Sample Program D, lines 68-76.

Write a System Sector

Writes a system sector (used in directory cylinder).

Entry Conditions:
A =54 (X'36')
HL = *pointer to the buffer containing the sector of data*
D = *cylinder to write*
E = *sector to write*
C = *logical drive number*

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
Only AF is altered by this SVC.

Example:
See Sample Program D, lines 94-104.

Write a Track

Writes an entire track of properly formatted data. The data format must conform to that described in the disk controller's reference manual. @WRTRK must always be preceded by @SLCT.

Entry Conditions:
A = 55 (X'37')
HL = *pointer to format data*
D = *track to write*
C = *logical drive number* (0-7)

Exit Conditions:
Success, Z flag set.
Failure, NZ flag set.
A = *error number*

General:
Only AF is altered by this SVC.

Numerical List of SVCs

Following is a numerical list of the SVCs:

Dec	Hex	Label	Function
0	00	@IPL	Reboot the system
1	01	@KEY	Scan *KI device, wait for character
2	02	@DSP	Display character at cursor, advance cursor
3	03	@GET	Get one byte from a logical device
4	04	@PUT	Write one byte to a logical device
5	05	@CTL	Make a control request to a logical device
6	06	@PRT	Send character to the line printer
7	07	@WHERE	Locate origin of CALL
8	08	@KBD	Scan keyboard and return
9	09	@KEYIN	Accept a line of input
10	0A	@DSPLY	Display a message line
11	0B	@LOGER	Issue a log message
12	0C	@LOGOT	Display and log a message
13	0D	@MSG	Message line handler
14	0E	@PRINT	Print a message line
15	0F	@VDCTL	Position/locate cursor, get/put character at cursor
16	10	@PAUSE	Suspend program execution
17	11	@PARAM	Parse an optional parameter string
18	12	@DATE	Get system date in the format MM/DD/YY
19	13	@TIME	Get system time in the format HH:MM:SS
20	14	@CHNIO	Pass control to the next module in a device chain
21	15	@ABORT	Load HL with X'FFFF' error and goto @ EXIT
22	16	@EXIT	Exit program and return to TRSDOS
23			Reserved for future use
24	18	@CMNDI	Entry to command interpreter with return to the system
25	19	@CMNDR	Entry to command interpreter with return to the user
26	1A	@ERROR	Entry to post an error message
27	1B	@DEBUG	Enter DEBUG
28	1C	@CKTSK	Check if task slot in use
29	1D	@ADTSK	Add an interrupt level task
30	1E	@RMTSK	Remove an interrupt level task
31	1F	@RPTSK	Replace the currently executing task vector
32	20	@KLTSK	Remove the currently executing task
33	21	@CKDRV	Check for drive availability
34	22	@DODIR	Do a directory display/buffer
35	23	@RAMDIR	Get directory record(s) or free space into RAM
36-39			Reserved for future use
40	28	@DCSTAT	Test if drive is assigned in DCT
41	29	@SLCT	Select a new drive
42	2A	@DCINIT	Initialize the FDC
43	2B	@DCRES	Reset the FDC
44	2C	@RSTOR	Issue FDC RESTORE command
45	2D	@STEPI	Issue FDC STEP IN command

Dec	Hex	Label	Function
46	2E	@SEEK	Seek a cylinder
47	2F	@RSLCT	Test if requested drive is busy
48	30	@RDHDR	Read a sector header
49	31	@RDSEC	Read a sector
50	32	@VRSEC	Verify a sector
51	33	@RDTRK	Read a track
52	34	@HDFMT	Hard disk format
53	35	@WRSEC	Write a sector
54	36	@WRSSC	Write a system sector
55	37	@WRTRK	Write a track
56	38	@RENAM	Rename a file
57	39	@REMOV	Remove a file or device
58	3A	@INIT	Open or initialize a file or device
59	3B	@OPEN	Open an existing file or device
60	3C	@CLOSE	Close a file or device
61	3D	@BKSP	Backspace one logical record
62	3E	@CKEOF	Check for end of file
63	3F	@LOC	Calculate the current logical record number
64	40	@LOF	Calculate the EOF logical record number
65	41	@PEOF	Position to the end of file
66	42	@POSN	Position a file to a logical record
67	43	@READ	Read a record from a file
68	44	@REW	Rewind a file to its beginning
69	45	@RREAD	Reread the current sector
70	46	@RWRIT	Rewrite the current sector
71	47	@SEEKSC	Seek a specified cylinder and sector
72	48	@SKIP	Skip the next record
73	49	@VER	Write a record to a file and verify
74	4A	@WEOF	Write end of file
75	4B	@WRITE	Write a record to a file
76	4C	@LOAD	Load a program file
77	4D	@RUN	Load and execute a program file
78	4E	@FSPEC	Fetch a file or device specification
79	4F	@FEXT	Set up a default file extension
80	50	@FNAME	Fetch filename/extension from directory
81	51	@GTDCT	Get Drive Code Table address
82	52	@GTDCB	Find specified or first free DCB
83	53	@GTMOD	Find specified memory module address
84			Reserved for future use
85	55	@RDSSC	Read a system sector
86			Reserved for future use
87	57	@DIRRD	Read directory record
88	58	@DIRWR	Write directory record
89			Reserved for future use
90	5A	@MUL8	Multiply 8-bit unsigned integers
91	5B	@MUL16	Multiply 16-bit by 8-bit unsigned integers
92			Reserved for future use
93	5D	@DIV8	Divide 8-bit unsigned integers
94	5E	@DIV16	Divide 16-bit by 8-bit unsigned integers
95			Reserved for future use
96	60	@DECHEX	Convert decimal ASCII to 16-bit binary value
97	61	@HEXDEC	Convert a number in HL to decimal ASCII

Dec	Hex	Label	Function
98	62	@HEX8	Convert a 1-byte number to hex ASCII
99	63	@HEX16	Convert a 2-byte number to hex ASCII
100	64	@HIGH$	Obtain or set the highest and lowest unused RAM addresses
101	65	@FLAGS	Point IY to the system flag table
102	66	@BANK	Check, set, or reset a 32K bank of memory
103	67	@BREAK	Set user or system break vector
104	68	@SOUND	Generate sound (tone and duration)
105-127			Reserved for future use.

Alphabetical List of SVCs

Following is an alphabetical list of the SVC labels and numbers:

Label	Dec	Hex
@ABORT	21	15
@ADTSK	29	1D
@BANK	102	66
@BKSP	61	3D
@BREAK	103	67
@CHNIO	20	14
@CKDRV	33	21
@CKEOF	62	3E
@CKTSK	28	1C
@CLOSE	60	3C
@CMNDI	24	18
@CMNDR	25	19
@CTL	5	5
@DATE	18	12
@DCINIT	42	2A
@DCRES	43	2B
@DCSTAT	40	28
@DEBUG	27	1B
@DECHEX	96	60
@DIRRD	87	57
@DIRWR	88	58
@DIV8	93	5D
@DIV16	94	5E
@DODIR	34	22
@DSP	2	2
@DSPLY	10	0A
@ERROR	26	1A
@EXIT	22	16
@FEXT	79	4F
@FLAGS	101	65
@FNAME	80	50
@FSPEC	78	4E
@GET	3	3
@GTDCB	82	52
@GTDCT	81	51
@GTMOD	83	53
@HDFMT	52	34
@HEXDEC	97	61
@HEX8	98	62
@HEX16	99	63
@HIGH$	100	64
@INIT	58	3A
@IPL	0	0
@KBD	8	8
@KEY	1	1
@KEYIN	9	9
@KLTSK	32	20
@LOAD	76	4C
@LOC	63	3F
@LOF	64	40
@LOGER	11	0B
@LOGOT	12	0C
@MSG	13	0D

Label	Dec	Hex
@MUL8	90	5A
@MUL16	91	5B
@OPEN	59	3B
@PARAM	17	11
@PAUSE	16	10
@PEOF	65	41
@POSN	66	42
@PRINT	14	0E
@PRT	6	6
@PUT	4	4
@RAMDIR	35	23
@RDHDR	48	30
@RDSEC	49	31
@RDSSC	85	55
@RDTRK	51	33
@READ	67	43
@REMOV	57	39
@RENAM	56	38
@REW	68	44
@RMTSK	30	1E
@RPTSK	31	1F
@RREAD	69	45
@RSLCT	47	2F
@RSTOR	44	2C
@RUN	77	4D
@RWRIT	70	46
@SEEK	46	2E
@SEEKSC	71	47
@SKIP	72	48
@SLCT	41	29
@SOUND	104	68
@STEPI	45	2D
@TIME	19	13
@VDCTL	15	0F
@VER	73	49
@VRSEC	50	32
@WEOF	74	4A
@WHERE	7	7
@WRITE	75	4B
@WRSEC	53	35
@WRSSC	54	36
@WRTRK	55	37

Sample Programs

The following sample programs use many of the supervisor calls described in this manual. These programs are not meant to be examples of the most efficient programming, but are designed to illustrate as many supervisor calls as possible.

Sample Program A

Source Line

```
;       This program asks the user whether to run a program
;       or debug it and executes the SVCs required to perform
;       the chosen action.

        PSECT   5000H           ;The program begins at x'5000'

;       Define the equates for the SVCs that will be used.

@DEBUG: EQU     27              ;Enter the debugger  (DEBUG)
@DSPLY: EQU     10              ;Display a message
@FSPEC: EQU     78              ;Verify a filespec or devspec and
                                ;load it into a File Control Block
@KEY:   EQU     1               ;Get a character from the keyboard
@LOAD:  EQU     76              ;Load a program into memory
@RUN:   EQU     77              ;Execute a program

MESS1:  DEFM    'Do you wish to RUN this Program or DEBUG it ?'
        DEFB    0AH             ;This moves the cursor to the next line
        DEFM    'Press <ENTER> to RUN or <BREAK> to DEBUG'
        DEFB    0DH             ;Terminate the message string

PROGRM: DEFM    'DIREX/CMD'     ;Sample program to debug or execute
        DEFB    0DH             ;Terminate the filespec

FCB1:   DEFS    32              ;File Control Block for the program

;       Get the File Control Block for the program 'DIREX/CMD'.

START:  LD      HL,PROGRM       ;Point at the filespec we want to
                                ;execute or load into memory
        LD      DE,FCB1         ;Point at the File Control Block
        LD      A,@FSPEC        ;Perform a validity check on the filespec
                                ;and copy the filespec into the FCB.
        RST     28H             ;Call the @FSPEC svc

        LD      HL,MESS1        ;Point at our prompting message
        LD      A,@DSPLY        ;and print it on the display
        RST     28H             ;Call the @DSPLY svc

        LD      A,@KEY          ;Get the reply from the keyboard
        RST     28H             ;Call the @KEY svc

        CP      0DH             ;Was the character an <ENTER>?
        JR      Z,RUNIT         ;If Z was set, then run the program

;       If it wasn't an <ENTER>, then we assume it was a <BREAK> and
;       load the program and enter the debugger.

        LD      DE,FCB1         ;Point at the File Control Block
        LD      A,@LOAD         ;and have this program loaded into memory
        RST     28H             ;Call the @LOAD svc

;       Note that this program must not be overwritten by the program
;       we are loading.  In this example, it is known that the program
;       we are loading starts at x'3000' and ends below x'5000'.

        LD      A,@DEBUG        ;Now invoke the system debugger, DEBUG
        RST     28H             ;Call the @DEBUG svc
                                ;Note that @DEBUG does not return

;       Execute the program

RUNIT:  LD      DE,FCB1         ;Point at the File Control Block
        LD      A,@RUN          ;Tell TRSDOS to load and execute the
                                ;program
        RST     28H             ;Call the @RUN svc
```

```
                                ;Note that @RUN returns only if it can't
                                ;find the program

;         Note that the program that is loaded by the @RUN svc must not
;         overwrite the File Control Block in this program.  In this case,
;         it is known that the program we are executing starts at x'3000'
;         and ends below the starting point of this program, x'5000'.

          END     START
```

```
;This program accepts numbers from the keyboard
;and uses them to demonstrate the
;arithmetic and numeric conversion SVCs.

;It also uses the sound function to produce a tone at the
;beginning of the program.

        PSECT   3000H

;       These are the SVCs used in this program.

@DECHEX:EQU     96              ;Convert decimal ASCII to binary
@DIV8:  EQU     93              ;Perform 8-bit division
@DIV16: EQU     94              ;Perform 16-bit division
@DSP:   EQU     2               ;Display a character
@DSPLY: EQU     10              ;Display a message
@EXIT:  EQU     22              ;Return to TRSDOS Ready or the caller
@HEX8:  EQU     98              ;Convert an 8-bit value to hex ASCII
@HEX16: EQU     99              ;Convert a 16-bit value to hex ASCII
@HEXDEC:EQU     97              ;Convert a binary value to Decimal ASCII
@KEY:   EQU     1               ;Read a character from *KI
@KEYIN: EQU     9               ;Accept an input line from *KI
@MUL8:  EQU     90              ;Perform 8-bit multiplication
@MUL16: EQU     91              ;Perform 16-bit multiplication
@SOUND: EQU     104             ;Produce a tone

;       Other equates.

NUM5:   EQU     5
NUM4:   EQU     4
NUM3:   EQU     3
NUM2:   EQU     2
NUM1:   EQU     1
BRK:    EQU     80H             ;Character code for <BREAK> key
CCC:    EQU     0DH             ;Next line position


;Perform a subroutine 2 times to display prompting messages, key in
;and display divisor and dividend, convert those numbers to
;binary for the divide, and position the cursor.

START:  LD      B,5AH           ;Make the longest, highest tone
        LD      A,@SOUND        ;Make the noise
        RST     28H
        CALL    KEYIN           ;Perform keyin subroutine for dividend
        LD      A,C
        LD      (DIVD1),A       ;Store the dividend in memory
        LD      HL,MESS9        ;Address of hex message
        CALL    DSPLAY          ;Display hex message
        LD      A,(DIVD1)       ;Get the divisor into C for conversion
        LD      C,A             ;from binary to hex
        CALL    HEX8            ;Convert the number to hex
        CALL    KEYIN           ;Perform subroutine for divisor
        LD      A,C
        LD      (DIVR1),A       ;Store the divisor in memory

;Now we are ready to perform the divide on the numbers entered.

        LD      C,A             ;Put the divisor back for the @DIV8 SVC
        LD      A,(DIVD1)       ;Get the dividend into E
        LD      E,A             ;for the @DIV8 SVC
        LD      A,@DIV8         ;Call the @DIV8 SVC
        RST     28H

;Now display the answer and the remainder in decimal.

        LD      (ANS1),A        ;Store the answer in memory
```

```
        LD      A,E             ;Get the remainder
        LD      (REM1),A        ;Store the remainder in memory
        LD      HL,MESS3        ;Load address of answer message
        CALL    DSPLAY          ;Display the message
        LD      A,(ANS1)        ;Get the answer into L for conversion
        LD      L,A             ;Number to convert
        LD      H,0             ;Put a 0 in the MSB
        CALL    HEXDEC          ;Perform subroutine to display decimal value
        LD      HL,MESS4        ;Address of remainder message
        CALL    DSPLAY          ;Display remainder message
        LD      A,(REM1)        ;Put remainder in A for hex conversion
        LD      L,A             ;Number to convert
        LD      H,0             ;Put 0 in the MSB
        CALL    HEXDEC          ;Display decimal value

;Now divide with a 16-bit dividend.

        LD      HL,MESS6        ;Address of 2nd dividend message
        CALL    DSPLAY          ;Display next message
        LD      A,@KEYIN        ;Key in up to 5 digits
        LD      HL,BUF6         ;Store the number
        LD      B,NUM5          ;Maximum length of number
        LD      C,0
        RST     28H
        LD      A,@DECHEX       ;Convert the number to binary
        RST     28H
        LD      (DIVD2),BC      ;Store the dividend
        LD      HL,MESS9        ;Address of hex message
        CALL    DSPLAY          ;Display hex message
        LD      DE,(DIVD2)      ;Put dividend into DE for conversion
        CALL    HEX16           ;Convert the number from binary to hex
        CALL    KEYIN           ;Key in divisor
        LD      A,C             ;Put the divisor into A
        LD      (DIVR1),A       ;Store the divisor in memory
        LD      HL,MESS3        ;Address of answer message
        CALL    DSPLAY          ;Display the message
        LD      HL,(DIVD2)      ;Put dividend into HL
        LD      A,(DIVR1)       ;Get divisor into C
        LD      C,A
        LD      A,@DIV16
        RST     28H
        LD      (REM1),A        ;Store the remainder
        LD      (ANS2),HL       ;Put the answer into HL
        CALL    HEXDEC          ;Display answer in decimal
        LD      HL,MESS4        ;Address of remainder message
        CALL    DSPLAY          ;Display remainder message
        LD      A,(REM1)        ;Get the remainder
        LD      L,A             ;into L
        LD      H,0             ;Put a 0 in MSB
        CALL    HEXDEC          ;Convert the remainder to decimal

;Now try some multiplication of 8 bits.

        LD      HL,MESS8        ;Address of MUL8 message
        CALL    DSPLAY          ;Display first multiplicand message
        LD      A,@KEYIN        ;Key in a 2-digit number
        LD      HL,BUF2         ;Put it here
        LD      B,NUM2          ;Maximum number of characters
        LD      C,0
        RST     28H
        LD      A,@DECHEX       ;Convert the number to binary for math
        RST     28H
        LD      (MCAND1),BC     ;Store the multiplicand
        LD      HL,MESS10       ;Address of MUL8 multiplier message
        CALL    DSPLAY          ;Display first multiplier message
        LD      A,@KEYIN        ;Key in the multiplier
        LD      HL,BUF2         ;Put it here
```

```
        LD      B,NUM1          ;Maximum number of characters
        LD      C,Ø
        RST     28H
        LD      A,@DECHEX       ;Convert the multiplier to binary for math
        RST     28H
        LD      (MIER1),BC      ;Store multiplier in memory
        LD      HL,MESS13       ;Address of multiplier message
        LD      A,@DSPLY        ;Display multiplier message
        RST     28H

;Now multiply the two numbers just entered.

        LD      A,(MCAND1)      ;Get the multiplicand into C
        LD      C,A
        LD      A,(MIER1)       ;Get the multiplier into E
        LD      E,A
        LD      A,@MUL8
        RST     28H
        LD      L,A             ;Put the product into L
        LD      H,Ø             ;Put Ø in the MSB
        CALL    HEXDEC          ;Convert the product to decimal

;Now multiply a 16-bit by an 8-bit.

        LD      HL,MESS11       ;Address of multiplicand message
        CALL    DSPLAY          ;Display 2nd multiplicand message
        LD      A,@KEYIN        ;Enter larger multiplicand
        LD      HL,BUF5         ;Put it here
        LD      B,NUM4          ;Maximum number of characters
        LD      C,Ø
        RST     28H
        LD      A,@DECHEX       ;Convert the number to binary for math
        RST     28H
        LD      (MCAND2),BC     ;Store the multiplicand in memory
        LD      HL,MESS12       ;Address of multiplier message
        CALL    DSPLAY          ;Display message
        LD      A,@KEYIN        ;Enter larger multiplier
        LD      HL,BUF3         ;Put it here
        LD      B,NUM2          ;Maximum number of characters
        LD      C,Ø
        RST     28H
        LD      A,@DECHEX       ;Convert the number to binary for math
        RST     28H
        LD      (MIER1),BC      ;Store the multiplier in memory
        LD      HL,MESS13       ;Address of product message
        LD      A,@DSPLY        ;Display the message
        RST     28H
        LD      HL,(MCAND2)     ;Put multiplicand into HL
        LD      A,(MIER1)       ;Get the multiplier into C
        LD      C,A
        LD      A,@MUL16        ;Multiply the two numbers
        RST     28H
        LD      H,L             ;Get the 2nd byte of the product into
                                ;H for conversion
        LD      L,A             ;Get the LSB into L for conversion
        LD      DE,BUF5         ;Convert the high-order byte to decimal
        LD      A,@HEXDEC       ;for the  display
        RST     28H
        LD      A,CCC           ;Tell the display when to stop
        LD      (DE),A
        LD      HL,BUF5
        LD      A,@DSPLY        ;Display the product
        RST     28H
        LD      HL,MESS14       ;Address of end message
        LD      A,@DSPLY        ;Display end message
        RST     28H
        LD      A,@KEY          ;Allow the user to enter any character
        RST     28H             ;or hit <BREAK>
```

```
        CP      BRK             ;Is it <BREAK>?
        JP      NZ,START        ;Yes, go back to beginning
        LD      A,@EXIT         ;No, exit the program
        RST     28H

;These are the subroutines used by the calls to
;display a message, key in a 3-digit number, and convert it
;from decimal to binary.

KEYIN:  LD      HL,MESS1
        CALL    DSPLAY          ;Display message
        LD      HL,BUF4         ;Put the number here
        LD      B,NUM3          ;Maximum number of characters
        LD      C,Ø
        LD      A,@KEYIN        ;Key in a number
        RST     28H
        LD      A,@DECHEX       ;Convert the number to binary
        RST     28H
        RET                     ;Return to next sequential instruction

;Display what was loaded into HL before the call.

DSPLAY: LD      A,@DSPLY    ;@DISPLAY SVC
        RST     28H
        DEC     HL              ;Set HL back to blank byte
        LD      B,(HL)          ;Load B with the number of bytes
DSPLYLP:LD      C,' '           ;Put a blank into C
        LD      A,@DSP          ;Display the blank
        RST     28H             ;until the correct number
        DJNZ    DSPLYLP         ;of blanks have been displayed
        RET                     ;Return to next instruction

;Convert 1 byte to hexadecimal.

HEX8:   LD      A,@HEX8         ;Convert 1 byte to hex ASCII
        LD      HL,BUF3         ;Put the converted value here
        RST     28H
        LD      A,CCC           ;Tell display when to stop
        LD      (HL),A          ;Put CCC at end of buffer
        LD      A,@DSPLY        ;Display the hex value
        LD      HL,BUF3
        RST     28H
        RET                     ;Return to next instruction

;Convert 2 bytes to hexadecimal.

HEX16:  LD      A,@HEX16        ;Convert a 2-byte number to hex ASCII
        LD      HL,BUF6         ;Put the converted value here
        RST     28H
        LD      A,CCC           ;CCC at end of buffer so display
        LD      (HL),A          ;knows when to stop
        LD      A,@DSPLY        ;Display the converted value
        LD      HL,BUF6         ;Address of converted value
        RST     28H
        RET                     ;Return to next instruction

;Convert from binary to decimal and display decimal value.

HEXDEC: LD      A,@HEXDEC       ;Convert from binary to decimal
        LD      DE,BUF5         ;Put converted value here
        RST     28H
        LD      A,CCC           ;CCC at end of buffer so display
        LD      (DE),A          ;knows when to stop
        LD      A,@DSPLY        ;Display the hex value
        LD      HL,BUF5         ;It's here
        RST     28H
        RET                     ;Return to next instruction
```

```
;These are the storage declarations.

BUF6:   DEFS    6
BUF5    DEFS    5
BUF4:   DEFS    4
BUF3:   DEFS    3
BUF2:   DEFS    2
DIVR1:  DEFB    0
DIVD1:  DEFB    0
ANS1:   DEFB    0
REM1:   DEFB    0
MCAND1: DEFB    0
MIER1:  DEFB    0
MCAND2: DEFW    0
DIVD2:  DEFW    0
ANS2:   DEFW    0

;Below are messages and prompting text used in the program.

        DEFB    13                ;Number of blanks to print after message 1
MESS1:  DEFM    'Enter a number (1-255).'
        DEFB    3                 ;Message-terminating character
        DEFB    21                ;Number of blanks to print after message 3
MESS3:  DEFM    'The answer is'
        DEFB    3                 ;Terminating character
        DEFB    18                ;Blanks after message
MESS4:  DEFM    'The remainder is'
        DEFB    3                 ;Terminating character
        DEFB    6                 ;Blanks after message
MESS6:  DEFM    'Enter a number (4369-65535).'
        DEFB    3                 ;Terminating character
        DEFB    15                ;Blanks after message
MESS8:  DEFM    'Enter a number (1-28).'
        DEFB    3                 ;Terminating character
        DEFB    16                ;Blanks after message
MESS9:  DEFM    'In hex ASCII, that is'
        DEFB    3                 ;Terminating character
        DEFB    17                ;Blanks after message
MESS10: DEFM    'Enter a number (1-9).'
        DEFB    3                 ;Terminating character
        DEFB    11                ;Blanks after message
MESS11: DEFM    'Enter a number (1-4100).'
        DEFB    3                 ;Terminating character
        DEFB    15                ;Blanks after message
MESS12: DEFM    'Enter a number (1-15).'
        DEFB    3                 ;Terminating character
MESS13: DEFM    'The product of those 2 numbers is '
        DEFB    3                 ;Terminating character
MESS14: DEFM    'Press <BREAK> to end or any other key to continue.'
        DEFB    0DH               ;Terminating character

        END     START
```

```
            Source Line

;           This program prompts for two filenames, opens the first
;           file, and creates the second.  Then the data in the first
;           file is copied to the second file.  While the Copy progresses,
;           the current record number is displayed in parentheses.

            PSECT   3ØØØH           ;This program starts at x'3ØØØ'

;           First, declare the equates for the SVCs we intend to use.
;           This is not mandatory, but it makes the program easier to follow.

@CLOSE:     EQU     6Ø              ;Close a file or device
@DIRRD:     EQU     87              ;Read a directory record
@DSP:       EQU     2               ;Display character at cursor
@DSPLY:     EQU     1Ø              ;Display a message
@ERROR:     EQU     26              ;Display an error message
@EXIT:      EQU     22              ;Exit and return to TRSDOS or the caller
@FEXT:      EQU     79              ;Add a default file extension
@FNAME:     EQU     8Ø              ;Fetch a filespec from the directory
@FSPEC:     EQU     78              ;Verify and load a filespec into the FCB
@HEXDEC:    EQU     97              ;Convert a binary value to decimal ASCII
@INIT:      EQU     58              ;Open an existing file or create a new file
@KBD:       EQU     8               ;Scan the keyboard for a character
@KEYIN:     EQU     9               ;Accept a line of text from the *KI device
@LOC:       EQU     63              ;Return the current logical record number
@OPEN:      EQU     59              ;Open an existing file
@READ:      EQU     67              ;Read a record from an open file
@REMOV:     EQU     57              ;Delete a file from disk
@VER:       EQU     73              ;Write a record to disk.  Does the same thing
                                    ;as @WRITE (Svc 75), but it also makes sure
                                    ;the written data is readable.

;           First, prompt for the source filespec using the @DSPLY svc.

BEGIN:      LD      HL,MESG1        ;Get the first message
            LD      A,@DSPLY        ;Display a line on the screen
            RST     28H             ;Call the @DSPLY svc

;           Now, read the filename from the keyboard using the @KEYIN svc.

            LD      HL,FILE1        ;Put the name of the 1st file here
            LD      B,24            ;Allow up to 24 characters
            LD      C,Ø             ;A zero is required by the svc
            LD      A,@KEYIN        ;Get a filename from the user
            RST     28H             ;Call the @KEYIN svc
            JP      C,QUIT          ;The user pressed <Break>
            JP      NZ,ERR          ;An Error occurred

            LD      A,B             ;Get the number of characters
            OR      A               ;See if that value was zero
            JR      Z,BEGIN         ;Nothing was entered, ask again

;           The user has typed something, so it must be checked for validity
;           using the @FSPEC svc.

            LD      HL,FILE1        ;Point at the text the user entered
            LD      DE,FCB1         ;Point at the File Control Block
                                    ;that is to be used for the source file.
            LD      A,@FSPEC        ;The @FSPEC svc will make sure the filename
                                    ;that is in buffer named "file1" is valid.
                                    ;If it is, it is copied into the File
                                    ;Control Block (FCB) to be used by the @OPEN
                                    ;or @INIT svc later on.
            RST     28H             ;Call the @FSPEC svc
            JR      Z,ASK2          ;The name for file 1 is ok, so skip this

;           At this point the filename specified for file 1 has been found
```

```
;       to be in an invalid format.  The following code will print the
;       error message.

        LD      HL,BADFIL       ;Point at the bad filename message
        LD      A,@DSPLY        ;Display it
        RST     28H             ;Call the @DSPLY svc
        JR      BEGIN           ;Start over

;       At this point, the source filename appears to be valid.
;       The code below asks for the second filename and checks it for
;       validity also.

ASK2:   LD      HL,MESG2        ;Prompt for the target filename
        LD      A,@DSPLY        ;Print that on the screen
        RST     28H             ;Call the @DSPLY svc
        LD      HL,FILE2        ;Put the name of the 2nd file here
        LD      B,24            ;Allow up to 24 characters
        LD      C,Ø             ;A zero is required by the svc
        LD      A,@KEYIN        ;Get a filename from the user
        RST     28H             ;Call the @KEYIN svc
        JP      C,QUIT          ;The user pressed <Break>
        JP      NZ,ERR          ;An Error occurred

        LD      A,B             ;Get the number of characters
        OR      A               ;See if that value was zero.
        JR      Z,ASK2          ;Nothing was entered, ask again

;       The user has typed something, so it must be checked for validity
;       using the @FSPEC svc.

        LD      HL,FILE2        ;Point at the text the user entered
        LD      DE,FCB2         ;Point at the File Control Block
        LD      A,@FSPEC        ;Check the name for validity
        RST     28H             ;Call the @FSPEC svc
        JR      Z,F2OK          ;The name for file 2 is ok, so skip this

;       The name for file 2 is invalid so print an error message

        LD      HL,BADFIL       ;Point at the bad filename message
        LD      A,@DSPLY        ;Display it
        RST     28H             ;Call the @DSPLY svc
        JR      BEGIN           ;Start over

;       Now we will attempt to add an extension to the target file
;       if the user did not specify one.  We use the extension that
;       was specified on the source file.  If it does
;       not have one, then we will not try to add one to the target file.

F2OK:   LD      HL,FCB1+1       ;Point at the source filename
                                ;We start with the second character since
                                ;the filename must be at least one character
FDIV:   LD      A,(HL)          ;Get a character from the filespec
        CP      '/'             ;Is the character the extension prefix?
        JR      Z,EXTN          ;Yes, this will be our default extension
        CP      ØDH             ;Have we reached the end of the filespec?
        JR      Z,NOEXT         ;Yes, there is no extension so don't add one
        CP      Ø3H             ;Test both terminators
        JR      Z,NOEXT
        INC     HL              ;Advance the pointer to the next character
        JR      FDIV            ;Keep looking

EXTN:   INC     HL              ;Advance pointer to first byte of extension
        LD      DE,FCB2         ;Point at FCB for the target file (file 2)
        LD      A,@FEXT         ;Add an extension if one is not present
        RST     28H             ;Call the @FEXT svc

;       Now we have two filenames.  First we will open the source file
;       to make sure it exists.
```

```

NOEXT:  LD      DE,FCB1         ;Point at the File Control Block for file1
        LD      HL,BUF1         ;Point at the system buffer.  This buffer
                                ;is used by the system to block data that
                                ;is written to disk and de-block data that
                                ;is read from disk when the Logical Record
                                ;Length of the file is not 256.  If it is
                                ;256, then this buffer is not used.
        LD      B,0             ;Use LRL 256 for now since we don't know
                                ;what to use yet.
        LD      A,@OPEN         ;Open the file
        RST     28H             ;Call the @OPEN svc
        JR      Z,SIZ           ;The file opened and is LRL 256.
        CP      42              ;Was the error a LRL Open Fault?
        JP      NZ,ERR          ;No, perhaps the file does not exist.

;       At this point, the file is open and we can now examine the
;       directory to find out what LRL it was created with so we can
;       use that value to make the copy.

SIZ:    LD      A,(FCB1+6)      ;Get the byte in the FCB which contains
                                ;the drive number the file is on
        AND     7               ;Erase all other information in that byte
        LD      C,A             ;Save that value here
        LD      A,(FCB1+7)      ;This reads the Directory Entry Code (DEC)
                                ;out of the FCB so we can use it
        LD      B,A             ;Store the DEC here
        PUSH    BC              ;Save that value for now
        LD      A,@CLOSE        ;We can close the source file for now
        RST     28H             ;Call the @CLOSE svc

        POP     BC              ;Get the DEC value back off the stack
        LD      A,@DIRRD        ;Read the directory record for that file
        RST     28H             ;Call the @DIRRD svc

        LD      IX,HL           ;Put the pointer to the directory record
        LD      A,(IX+4)        ;here and read the DIR+4 entry which
                                ;contains the LRL of the source file.
        LD      (LRL),A         ;Save that value

;       Before we go any further, we should check to see if the target file
;       already exists.

        LD      DE,COPY         ;First, make a copy of the FCB
        LD      HL,FCB2         ;in case we have to delete a file
        LD      BC,32           ;Move the entire block
        LDIR

        LD      DE,FCB2         ;Point at the target File Control Block
        LD      HL,BUF2         ;Use this as the buffer for now
        LD      B,0             ;Use LRL 256 for now
        LD      A,@OPEN         ;Open it and see if it is there
        RST     28H             ;Call the @OPEN svc
        JR      Z,EXISTS        ;The file already exists, better ask
        CP      42              ;Was the error a LRL mismatch?
        JR      NZ,NOFILE       ;No, so the file does not exist.

EXISTS: LD      HL,FEXST        ;Point at a prompt asking if it is ok
                                ;to erase the file that already exists
        LD      A,@DSPLY        ;Print that message
        RST     28H             ;Call the @DSPLY svc

WAIT:   LD      A,@KBD          ;Wait for the user to type Y or N
        RST     28H             ;Call the @KBD svc
        JR      NZ,WAIT         ;Loop until something is typed

        CP      'Y'             ;Was a 'Y' typed?
        JR      Z,KILLIT        ;Then kill the file
```

```
                CP      'y'             ;Check for lowercase too
                JR      Z,KILLIT
                CP      'N'             ;Do they want to leave the file alone?
                JR      Z,SHUT          ;No, just close the file and quit
                CP      'n'             ;Was it a lowercase 'N'?
                JR      NZ,WAIT         ;No, loop until we see something we like

        SHUT:   LD      DE,FCB2         ;Close the target file
                LD      A,@CLOSE
                RST     28H             ;Call the @CLOSE svc
                JP      QUIT            ;Exit to TRSDOS

        ;       At this point, we have been given the OK to delete the file
        ;       that has the same name as the target file.

        KILLIT: LD      C,ØDH           ;First move display to a new line
                LD      A,@DSP          ;Display an <Enter>
                RST     28H             ;Call the @DSP svc

                LD      DE,FCB2         ;Point at the target file's FCB
                LD      A,@REMOV        ;Delete the file from disk
                RST     28H             ;Call the @REMOV svc.  (This is the same
                                        ;as the @KILL call on other TRSDOS systems.)
                JP      NZ,ERR          ;An error occurred, print it and quit
                                        ;Note that after a @REMOV succeeds,
                                        ;the filespec is removed from the FCB.
                                        ;So we have to keep a copy around
                                        ;in case we need it.
                LD      HL,COPY         ;Get the copy
                LD      DE,FCB2         ;Put it here
                LD      BC,32           ;Move up to 32 bytes
                LDIR                    ;Copy the FCB so we can continue

        ;       Now we know what Logical Record Length (LRL) to use in the
        ;       copy, so we can open the source file and create the target file
        ;       with the correct record lengths.

        NOFILE: LD      HL,FCB1         ;Point at the filename in the FCB
                LD      A,@DSPLY        ;Print that name
                RST     28H             ;Call the @DSPLY svc
                LD      HL,SPACES       ;Point at some spaces
                LD      A,@DSPLY        ;Space over a few places on the screen
                RST     28H             ;Call the @DSPLY svc

                LD      DE,FCB1         ;Point at File Control Block for source file
                LD      HL,BUF1         ;Put data in this
                LD      A,(LRL)         ;Read the Logical Record Length
                LD      B,A             ;Load the Logical Record Length
                LD      A,@OPEN         ;Open the source file
                RST     28H             ;Call the @OPEN svc
                JP      NZ,ERR          ;Open failed

                LD      HL,ARROW        ;Point at the arrow text
                LD      A,@DSPLY        ;Print that to show the direction of copy
                RST     28H             ;Call the @DSPLY svc

                LD      DE,FCB2         ;Point at File Control Block for target file
                LD      A,(LRL)         ;Get the Logical Record Length
                CP      Ø               ;Is the LRL 256?
                JR      Z,LRL256        ;Then we do something special
                LD      HL,BUF2         ;Use a different buffer for target file
                JR      LRLCOM          ;Jump to common code
        LRL256: LD      HL,BUF1         ;We use the same buffer when the LRL is 256
                                        ;since there is no need to block and de-block
                                        ;the data.
        LRLCOM: LD      B,A             ;Load the Logical Record Length
                LD      A,@INIT         ;Open the target file
```

```
          RST     28H             ;Call the @INIT svc
          JR      NZ,ERR          ;Init failed

          LD      DE,FILE2        ;We are going to get the filename for
                                  ;the target file from the system
                                  ;instead of using the one we have.  The
                                  ;reason for this is that the system will
                                  ;append the drive number to the filename
                                  ;if one was not specified.
          LD      A,(FCB2+7)      ;Get the Directory Entry Code for the file
          LD      B,A             ;Put the DEC here
          LD      A,(FCB2+6)      ;Get the Drive Number from the FCB
          AND     7               ;Lose all data except the drive number
          LD      C,A             ;Store drive number here
          LD      A,@FNAME        ;Have the system produce a filespec
          RST     28H             ;Call the @FNAME svc
          LD      HL,FILE2        ;Now point at the filespec produced
          LD      A,@DSPLY        ;and print it out
          RST     28H             ;Call the @DSPLY svc

          LD      HL,SPACES       ;Space over a few more places
          LD      A,@DSPLY        ;so the display will look neat
          RST     28H             ;Call the @DSPLY svc

;         At this point, both files are open and ready to be used.
;         The following code reads a record from the source file
;         and writes it to the target file.  This is done until an
;         end of file is encountered.

LOOP:     LD      DE,FCB1         ;Point at file 1 (source file)
          LD      HL,BUFFER       ;Put data here
          LD      A,@READ         ;Read a record from the source file
          RST     28H             ;Call the @READ svc
          JR      NZ,EOF          ;Jump if the eof has been reached
          LD      DE,FCB2         ;Point at file 2 (target file)

;         Before writing the record, display the record number, which
;         is obtained from the @LOC svc.

          LD      A,@LOC          ;Get the current record number
          RST     28H             ;Call the @LOC svc

          PUSH    BC              ;Get the current record number
          POP     HL              ;and put it in register HL
          LD      DE,LOCMSG+1     ;Store the result here.
          LD      A,@HEXDEC       ;Convert binary to ASCII in decimal format
          RST     28H             ;Call the @HEXDEC svc

          LD      A,' '           ;Get a blank
          LD      HL,LOCMSG       ;Look at the front of the buffer
EDIT:     CP      (HL)            ;Is the character a blank?
          JR      NZ,NUMBR        ;A number has been found
          INC     HL              ;Advance the pointer
          JR      EDIT            ;Loop until we find a number

NUMBR:    DEC     HL              ;Back up one position
          LD      A,'('           ;Get the character we want to insert
          LD      (HL),A          ;Store that character.
                                  ;The buffer now contains
                                  ;<none or more spaces>(record number)
                                  ;<7 left-cursor characters><etx>
          LD      HL,LOCMSG       ;Point at this text
          LD      A,@DSPLY        ;and display it on the screen
          RST     28H             ;Call the @DSPLY svc

;         Now write the record to the target file.

          LD      DE,FCB2         ;Point at the FCB for the target file
```

```
        LD      HL,BUFFER       ;Point at the data read from file 1
        LD      A,@VER          ;Write a record to the target file
                                ;The @VER does the same thing as the
                                ;@WRITE svc, only it also checks the
                                ;data to make sure it is readable.
        RST     28H             ;Call the @VER svc
        JR      NZ,ERR          ;An error occurred on write; possibly
                                ;the disk is full.
        JR      LOOP            ;Loop until an error occurs.
;       This code checks the error to make sure it was an end of file
;       condition and, if so, closes the source & target files.
EOF:    CP      28              ;Was it an end of file encountered?
        JR      Z,EOFYES        ;Yes, close the file
        CP      29              ;Was it "Record number out of range"?
        JR      NZ,ERR          ;No, must be some other error
;       It is possible to get Error 29 if the file being copied has
;       an EOF that is not a multiple of the file's LRL
EOFYES: LD      DE,FCB1         ;Point at file 1 (source file)
        LD      A,@CLOSE        ;Close the file
        RST     28H             ;Call the @CLOSE svc
        JR      NZ,ERR          ;An error occurred, abort
        LD      DE,FCB2         ;Point at file 2 (target file)
        LD      A,@CLOSE        ;Close it also
        RST     28H             ;Call the @CLOSE svc
        JR      NZ,ERR          ;An error occurred, abort
        LD      HL,OK           ;Print a message saying the copy is done
        LD      A,@DSPLY
        RST     28H             ;Call the @DSPLY svc
QUIT:   LD      A,@EXIT         ;Exit to TRSDOS or the calling program
        RST     28H             ;Call the @EXIT svc
;       The @EXIT svc does not return.
ERR:    OR      Ø4ØH            ;Turn on bit 6, which
                                ;will cause the @ERROR svc to print
                                ;the short error message. Bit 7
                                ;is not set, which instructs the @ERROR
                                ;to abort this program and return to
                                ;TRSDOS Ready.
        LD      C,A             ;Put error code & flags in register C
        LD      A,@ERROR        ;Call the system error displayer
        RST     28H             ;Call the @ERROR svc
;       Because bit 7 is not set, the @ERROR svc will not return.
;       Storage Declaration
SPACES: DEFM    '  '            ;ASCII Space char.for display formatting
        DEFB    3
ARROW:  DEFM    '=>  '          ;Arrow for display shows data direction
        DEFB    3
OK:     DEFB    1Ø%25           ;Advance cursor 1Ø spaces without erasing
        DEFM    '[Ok]'          ;Used to indicate the Copy is complete
        DEFB    ØDH             ;Terminated with an <Enter>
MESG1:  DEFM    'Copy Filespec >'
        DEFB    3
MESG2:  DEFM    'To Filespec >'
        DEFB    3
FEXST:  DEFM    'Destination File Already Exists - Ok to Delete it (Y/N) ?'
        DEFB    3
```

```
BADFIL: DEFM    'Invalid Filename - Try Again'
        DEFB    0DH
LOCMSG: DEFM    ' 12345)'         ;This will be used in building the LOC
                                  ;Display will appear as (d) to (ddddd).
        DEFB    7%24              ;Backspace without erasing
        DEFB    3                 ;Etx, used to get the @DSPLY svc to stop

FILE1:  DEFS    32                ;User Text Originally placed here
FILE2:  DEFS    32                ;Target Filename goes here
FCB1:   DEFS    32                ;32 bytes for the File Control Block
FCB2:   DEFS    32                ;32 bytes for the File Control Block
COPY:   DEFS    32                ;An extra copy of the target FCB goes here
LRL:    DEFB    0                 ;The Logical Record Length of the source
                                  ;file will be stored here
BUF1:   DEFS    256               ;System buffer for File 1
BUF2:   DEFS    256               ;System buffer for File 2
BUFFER: DEFS    256               ;Data buffer for both files

        END     BEGIN             ;"begin" is the starting address
```

Sample Program D

```
                Source Line

;               This program will read a sector from the disk in Drive 0
;               and will write it to a disk in Drive 1.  The disk in Drive 1
;               must be formatted, but should not have anything important on
;               it.  This program makes an assumption that the directory is
;               located on cylinder 20 (x'14').

                PSECT   3000H           ;This program begins at x'3000'.

;               Define the equates for the SVCs that will be used.

@ABORT:         EQU     21              ;Abort and return to TRSDOS
@CKDRV:         EQU     33              ;Test to see if a drive is ready
@DCSTAT:        EQU     40              ;Verify that a drive is defined in the DCT
@ERROR:         EQU     26              ;Display an error message
@EXIT:          EQU     22              ;Return to TRSDOS or the calling program
@RDSEC:         EQU     49              ;Read a sector
@RDSSC:         EQU     85              ;Read a system sector
@WRSEC:         EQU     53              ;Write a sector
@WRSSC:         EQU     54              ;Write a system sector

;               Other Equates

SYSSEC:         EQU     1400H           ;The system sector is Cylinder 20, Sector 0
USRSEC:         EQU     0000H           ;The regular sector is Cylinder 0, Sector 0

;               First, test the target drive and make sure it is defined.

START:          LD      C,1             ;Select Drive 1
                LD      A,@DCSTAT       ;Ask if the drive is listed in the DCT
                RST     28H             ;Call the @DCSTAT svc
                JR      NZ,ERROR        ;If NZ, then the drive is not defined
                                        ;and we will abort execution.

;               Now, test and make sure the target drive contains a formatted
;               disk and is write-enabled.

                LD      C,1             ;Select Drive 1
                LD      A,@CKDRV        ;Test to see if the disk is formatted
                                        ;and is write-enabled.  Note that the
                                        ;disk must be formatted by TRSDOS 6.x
                                        ;or by LDOS 5.1.x to be considered
                                        ;"formatted" by this svc.
                RST     28H             ;Call the @CKDRV svc
                LD      A,8             ;This will become the error number if the
                                        ;drive was not ready.  This is done
                                        ;because the @CKDRV svc does not return error
                                        ;codes.
                JR      NZ,ERROR        ;The drive is not ready
                LD      A,15            ;This will become the error number if the
                                        ;drive is ready and is write-protected.
                                        ;As above, this is done because @CKDRV does
                                        ;not return error messages.
                JR      C,ERROR         ;The disk is formatted, but it is
                                        ;write-protected.  In either case, abort.

;               Now that we know the target drive is ready, read a sector
;               from the source drive and write it to the target drive (Drive 1).

                LD      C,0             ;Select Drive 0
                LD      DE,USRSEC       ;Read the first sector on the disk,
                                        ;Cylinder 0, Sector 0.
                LD      HL,BUFF         ;Point to a buffer which will hold the sector
                LD      A,@RDSEC        ;Read a non-system sector
                RST     28H             ;Call the @RDSEC svc
                JR      NZ,ERROR        ;If NZ, an error occurred, so abort

```

```
;        Now, write the sector to the target drive.

         LD      C,1                 ;Select Drive 1
         LD      DE,USRSEC           ;Write the sector to Cylinder Ø, Sector Ø
                                     ;on Drive 1
         LD      HL,BUFF             ;Point to the buffer containing the sector
         LD      A,@WRSEC            ;Write the sector to disk
         RST     28H                 ;Call the @WRSEC svc
         JR      NZ,ERROR            ;If NZ, an error occurred, so abort

;        Now we will read a system sector from Drive Ø and write it on
;        drive 1.  The difference between a system sector and a non-system
;        sector is that the Data Address Marks (DAM) are different.  These
;        were written to the disk when it was formatted.  TRSDOS 6.x uses
;        these as an extra check to make sure that a write of user data
;        does not accidentally get placed over a sector containing system
;        data.  All of the sectors in the directory cylinder are marked
;        as system sectors.

         LD      C,Ø                 ;Select Drive Ø
         LD      DE,SYSSEC           ;Read Cylinder 2Ø, Sector Ø
         LD      HL,BUFF             ;Store the sector at this address
         LD      A,@RDSSC            ;Read a system sector
         RST     28H                 ;Call the @RDSSC svc
         JR      NZ,ERROR            ;An error occurred, so abort

;        Now write the sector to the target drive as a system sector.
;        There is no requirement that a sector must be placed at the
;        same cylinder and sector location as it was read from, but
;        for simplicity, we are doing that.

         LD      C,1                 ;Select Drive 1
         LD      DE,SYSSEC           ;Write Cylinder 2Ø, Sector Ø
         LD      HL,BUFF             ;Point to the data to be written
         LD      A,@WRSSC            ;Write a system sector
         RST     28H                 ;Call the @WRSSC svc
         JR      NZ,ERROR            ;An error occurred, so abort

         LD      A,@EXIT             ;Return to TRSDOS or the calling program
         RST     28H                 ;Call the @EXIT svc

;        This routine displays an error message if anything goes wrong.
;        Note that @CKDRV does not return an error message, so @ERROR
;        cannot be used for it without some manipulation.

ERROR:   OR      ØCØH                ;Set bit 7
         LD      C,A                 ;Load error number into register C
         LD      A,@ERROR            ;This will display the error message
                                     ;and return to the calling program
         RST     28H                 ;Call the @ERROR svc

         LD      A,@ABORT            ;Now, force an abort. This will return
                                     ;to TRSDOS Ready and will abort any
                                     ;JCL file that is currently executing
         RST     28H                 ;Call the @ABORT svc

BUFF:    DEFS    256                 ;256-byte buffer to store the sector that
                                     ;is read and then written

         END     START
```

Sample Program E

```
                 Source Line
;        This program displays the filenames of the disk in
;        Drive Ø three different ways.
         PSECT   3ØØØH           ;Program begins at x'3ØØØ'
;        First, declare the equates for the SVCs we intend to use.
;        This is not mandatory, but it makes the program easier to follow.
@CMNDI:  EQU     24              ;Execute a TRSDOS command and return
                                 ;to TRSDOS Ready
@CMNDR:  EQU     25              ;Execute a TRSDOS command and return
                                 ;to the calling program
@DODIR:  EQU     34              ;Display visible filenames on the
                                 ;specified disk drive
;        First, pass a "DIR :Ø" command to the system.  TRSDOS will
;        execute this command and then return to this program.
START:   LD      HL,DIRØ         ;Point at command we want to execute
         LD      A,@CMNDR        ;Execute the specified command and return
         RST     28H             ;Call the @CMNDR svc
;        You may have noticed that the DIR displayed the files, but that
;        they were not sorted alphabetically.  This is because the DIR
;        command will not use memory above x'3ØØØ' when it is invoked with
;        a @CMNDR svc.  This prevents the DIR command from performing a
;        sort of the filenames.
;        Now do a directory command using the @DODIR svc.
         LD      B,Ø             ;Use Function Ø which displays all
                                 ;visible files in the directory.
         LD      C,Ø             ;Put source drive number in register C
         LD      A,@DODIR        ;The filenames will be read from the
                                 ;directory and displayed in the
                                 ;order they appear in the directory.
         RST     28H             ;Call the @DODIR svc
;        Now pass a "DIR :Ø" command to the system.  This time
;        the command will be executed and then TRSDOS will not return
;        to this program, but will return to TRSDOS Ready.
         LD      HL,DIRØ         ;Point at the command we want performed
         LD      A,@CMNDI        ;and execute it, but don't return to
                                 ;this program.
         RST     28H             ;Call the @CMNDI svc
                                 ;This svc returns to TRSDOS Ready.
;        Note that when the library command DIR is performed this time,
;        the display of files is sorted.  This is because DIR determines
;        that it was invoked with a @CMNDI svc, and it will not return
;        to the calling program.  Therefore, DIR is free to use the
;        memory above x'3ØØØ' to perform the sort of the filenames in
;        the directory.
;        Constants
DIRØ:    DEFM    'DIR :Ø'        ;This command is passed to TRSDOS
                                 ;via the @CMNDR and @CMNDI SVCs.
         DEFB    ØDH             ;It must be terminated with an <ENTER>.
         END     START
```

Sample Program F

```
        Source Line

;       This program adds to the system task scheduler a task
;       which displays the date and a running count of the number
;       of times the task has been executed.
;       For simplicity, the program tries to use task slot 0.
;       If it is already in use, it assumes that the task using that
;       slot is this program, and it kills the task. It then tries to
;       recover the memory used by the task in high memory.
;       If the task slot is not in use, the task is placed in high memory,
;       and the address of the task is passed to the task scheduler.
;       The first time you run this program it adds the task, and the
;       next time you run this program, it removes the task.

        PSECT   3000H           ;This program starts at x'3000'

;       First, declare the equates for the SVCs we intend to use.
;       This is not mandatory, but it makes the program easier to follow.

@ADTSK: EQU     29              ;Add a task entry to the scheduler
@CKTSK: EQU     28              ;Check to see if a task slot is in use
@DATE:  EQU     18              ;Return the date in ASCII format
@DSPLY: EQU     10              ;Display a message
@EXIT:  EQU     22              ;Return to TRSDOS Ready or the caller
@GTMOD: EQU     83              ;Locate a memory module
@HEXDEC:EQU     97              ;Convert a binary value to decimal ASCII
@HIGH$: EQU     100             ;Read or modify HIGH$ or LOW$
@RMTSK: EQU     30              ;Remove a task entry from the scheduler
@VDCTL: EQU     15              ;Perform video operations
@WHERE: EQU     7               ;Find out where the program counter is
                                ;when this SVC is executed.  This is
                                ;useful in relocatable code that must
                                ;make absolute address references to
                                ;call subroutines or modify data.


;       Below we will define a macro to simulate a call relative
;       instruction.  Since the task must be able to run no matter
;       where it is placed, it must use relative jumps and calls.
;       The Z80 instruction set has a jump relative (JR), but does
;       not have a call relative instruction.  This can be simulated
;       using the @WHERE SVC, which returns the address of the caller
;       in a register.  This address can be adjusted and placed on
;       the stack as a return address. Then a jump relative can be used
;       to reach the subroutine.

CALLR:  MACRO   #1              ;#1 will be the address you want to call
        PUSH    HL              ;Save the registers we damage
        PUSH    BC              ;Save it
        PUSH    AF              ;Save it
        LD      A,@WHERE        ;Get our current address
        RST     28H             ;Call the @WHERE svc
        LD      BC,3+1+1+1+1+2  ;Get the lengths of the instructions after
                                ;the SVC.  This will allow the subroutine
                                ;to return to the correct address.
        ADD     HL,BC           ;Add that offset to where we are
        POP     AF              ;Put stack back
        POP     BC              ;Restore registers
        EX      (SP),HL         ;Put return address on stack and restore HL
        JR      #1              ;Jump to the subroutine
        ENDM                    ;End of the macro


;       This is the main program.  It loads at x'3000'.  It decides
;       if it needs to add or remove the task in the scheduler tables.
;       If it adds the task, it moves a copy to the top of memory and
;       protects it, and adds a task entry to the scheduler.
;       If it is removing a task, it kills the entry in the scheduler
```

```
;       tables, and then attempts to recover the memory used by the task.

BEGIN:  LD      C,0                 ;First, we will test slot 0
        LD      A,@CKTSK            ;to see if anyone is using it
        RST     28H                 ;Call the @CKTSK svc
        JR      NZ,KILLIT           ;There is a task using slot 0, kill it

;       At this point, we want to add a task to high memory.
;       First we find the value for HIGH$ and put a copy of the
;       task there.  Then we protect the task by moving HIGH$ below
;       the new task.

        LD      HL,0                ;First, get the value of HIGH$
        LD      B,H                 ;Read HIGH$
        LD      A,@HIGH$
        RST     28H                 ;Call the @HIGH$ svc
        LD      (ENDADD),HL         ;Save this value as the last address
                                    ;that the task will be stored in once it
                                    ;is moved to high memory

        LD      DE,HL               ;Put that value here
        LD      HL,MODEND-1         ;Point at the end of the module
        LD      BC,MODEND-MODULE;Move the module from where it is
                                    ;right now to a position below HIGH$
        LDDR                        ;Do the copy

        LD      HL,DE               ;Now protect the module using HIGH$
        LD      B,0                 ;Update HIGH$
        LD      A,@HIGH$
        RST     28H                 ;Call the @HIGH$ svc

;       Now we need to load the TCB entry in the module with the address
;       of the first instruction to be executed.

        LD      IX,HL               ;IX now points at memory header
        LD      BC,ENTRY-MODULE+1           ;Get the offset into the module
                                            ;of the first instruction
        ADD     HL,BC               ;HL now contains the actual starting address
        LD      (IX+(1+MODTCB-MODULE)),L            ;Store LSB of the address
        LD      (IX+1+(1+MODTCB-MODULE)),H          ;Store MSB of the address

;       Now the task is ready to run.  We now add the entry to the task
;       scheduler table.

        LD      BC,MODTCB-MODULE+1          ;Get offset into the
                                            ;module of the TCB word
        PUSH    IX                  ;Get a copy of the base address
        POP     HL                  ;Put base address here
        ADD     HL,BC               ;Now HL points at TCB address
        LD      DE,HL               ;Put that value in DE
        LD      C,0                 ;Add this entry to task slot 0
        LD      A,@ADTSK            ;Add this task, to be run every 266.67 msec
        RST     28H                 ;Call the @ADTSK svc

;       The main program has now done its work and can exit.

        LD      HL,ADDED            ;Point at a message saying what was done
        LD      A,@DSPLY            ;and print it
        RST     28H                 ;Call the @DSPLY svc

        LD      A,@EXIT             ;Now exit
        RST     28H                 ;Call the @EXIT svc

;       This SVC does not return.


;       This part of the code removes the task from the scheduler
;       tables and then attempts to recover the memory that was used
```

```
;       by the task in high memory.  If another high memory module
;       was added AFTER this task was added, then the memory that
;       was used by this task cannot be recovered.

KILLIT: LD      C,0             ;We want to remove the task in slot 0
        LD      A,@RMTSK
        RST     28H             ;Call the @RMTSK svc

;       At this point, the task is no longer called by the operating
;       system.  Now we want to determine if we can
;       reclaim the memory it was using.

        LD      DE,MODNAM       ;Point at the name of the module
        LD      A,@GTMOD        ;Look for a module with that name
        RST     28H             ;Call the @GTMOD svc
        JR      NZ,CANT         ;If NZ is set, then we killed some other
                                ;task that was using slot 0.  Oops.
                                ;In that case, just stop and don't do any
                                ;more damage.
        LD      IX,HL           ;Set IX to point to the module.
        LD      B,0             ;Read the current value of HIGH$
        LD      HL,0            ;to see if this is the first program in
                                ;high memory
        LD      A,@HIGH$        ;If it is, then we can recover the space
        RST     28H             ;Call the @HIGH$ svc
        INC     HL              ;Move HIGH$ up by one byte
        PUSH    IX              ;Take the address of our module
        POP     DE              ;and store it here
        XOR     A               ;Compare these
        SBC     HL,DE           ;Are they the same?
        JR      NZ,CANT         ;No, the high memory module can't be removed

;       At this point, we know it is ok to reclaim the memory used by the
;       high memory task.

        LD      HL,(IX+2)       ;Read the end of module value out of the
                                ;header information
        LD      B,0             ;Update the HIGH$ value
        LD      A,@HIGH$
        RST     28H             ;Call the @HIGH$ svc

        LD      HL,OK           ;Point to a message saying all is well
        LD      A,@DSPLY        ;and print it
        RST     28H             ;Call the @DSPLY svc

        LD      A,@EXIT         ;Exit the main program
        RST     28H             ;Call the @EXIT svc


;       Here we will display a message saying we removed the task from
;       the scheduler table, but we cannot reclaim the memory that was
;       used.

CANT:   LD      HL,RECLM        ;Point to the message
        LD      A,@DSPLY        ;and display it
        RST     28H             ;Call the @DSPLY svc

        LD      A,@EXIT         ;Now exit
        RST     28H             ;Call the @EXIT svc


;       Messages

ADDED:  DEFM    'Task placed in high memory and scheduled.'
        DEFB    0DH
OK:     DEFM    'Task removed from scheduler table and memory reclaimed.'
        DEFB    0DH
RECLM:  DEFM    'Task removed from scheduler table, but memory could not '
```

```
                DEFM    'be recovered.
                DEFB    0DH

;               The Task begins at this point.  This part of the program loads
;               in low memory but is relocated to a point just below HIGH$.

;               This is the Memory Header Block.  This block of data allows
;               the system to locate this module in memory by name,
;               using the @GTMOD svc.

MODULE: JR      ENTRY                   ;Jump (relative) to the starting address
ENDADD: DEFW    0                       ;The highest address in the program.
                                        ;This value is patched in before the program
                                        ;is relocated.  This will be used
                                        ;later in recovering the memory used by
                                        ;this task.
        DEFB    MODTCB-MODNAM           ;Number of bytes in the name field below.
MODNAM: DEFM    'UPTIME'                ;This is the name of the module and is
                                        ;used to identify the module.
MODTCB: DEFW    0                       ;Actual address to start execution.  This
                                        ;value is patched in after the program is
                                        ;relocated.
        DEFW    0                       ;Spare system pointer - RESERVED

;               This area contains data used by the task.  It is addressed using
;               the IX register which points to the task when it is executed.

COUNTER:DEFW    0                       ;Count of how many times we have run
DATBUF: DEFS    9                       ;The date is stored here

;               This is the actual task.
;               On entry to the task, IX points at the Task Control Block (TCB),
;               which in this program is the label 'MODTCB'.  All data is
;               referenced by indexing from that address.


ENTRY:  PUSH    IY                      ;Save this register.  It is not saved by
                                        ;the Task Scheduler, and we use it.
                                        ;Registers AF, BC, DE, and HL are saved

;               Now we will read the current date.

        LD      HL,IX                   ;Get a copy of the index pointer
        LD      BC,DATBUF-MODTCB        ;Get the offset needed to access the date
        ADD     HL,BC                   ;Now we have a pointer to the date

        PUSH    IX                      ;Save the pointer to the start of the task
        PUSH    HL                      ;Save a copy of that pointer
        LD      A,@DATE                 ;Ask the system what the date is
        RST     28H                     ;Call the @DATE svc

        LD      (HL),0                  ;Terminate the date string

        POP     DE                      ;Put pointer to the date here
        PUSH    DE                      ;We will use this pointer later on
        LD      HL,0028H                ;Put the cursor on the top line,
                                        ;specified in register HL
                                        ;at the 41st position on the screen
        CALLR   WRITE                   ;Write the message at the position
+       PUSH    HL                      ;Save the registers we damage
+       PUSH    BC                      ;Save it
+       PUSH    AF                      ;Save it
+       LD      A,@WHERE                ;Get our current address
+       RST     28H                     ;Call the @WHERE svc
+       LD      BC,3+1+1+1+1+2          ;Get the lengths of the instructions after
+                                       ;the SVC.  This will allow the subroutine
+                                       ;to return to the correct address.
```

```
        ADD     HL,BC           ;Add that offset to where we are
        POP     AF              ;Put stack back
        POP     BC              ;Restore registers
        EX      (SP),HL         ;Put return address on stack and restore HL
        JR      WRITE           ;Jump to the subroutine
                                ;Note that the above was actually a macro
                                ;which performs a relative call.

;       This part of the task displays a count of the number of times
;       the task has been executed.

        POP     DE              ;Get the pointer to DATBUF back
        POP     IX              ;Get the pointer to the beginning of
                                ;this task
        PUSH    DE              ;Save the pointer to DATBUF again
        LD      BC,COUNTER-MODTCB       ;Get the offset to our data
                                        ;area
        LD      HL,IX           ;Put a copy of the base address in HL
        ADD     HL,BC           ;Add offset.  Now HL points to COUNTER:
        LD      IY,HL           ;Put the pointer to COUNTER in IY
        LD      L,(IY)          ;Get LSB of the counter
        LD      H,(IY+1)        ;Get MSB of the counter
        INC     HL              ;Increment the number of times we have run
        LD      (IY),L          ;Store the LSB of the counter
        LD      (IY+1),H        ;Store the MSB of the counter

        LD      A,@HEXDEC       ;Convert the count to decimal
        RST     28H             ;Call the @HEXDEC svc

        XOR     A               ;Get a zero
        LD      (DE),A          ;Terminate the count string

        POP     DE              ;Put pointer to date here
        LD      HL,0036H        ;Put the cursor on the top line,
                                ;specified in register HL
                                ;at the 55th position on the screen
        CALLR   WRITE           ;Write the message at the position
        PUSH    HL              ;Save the registers we damage
        PUSH    BC              ;Save it
        PUSH    AF              ;Save it
        LD      A,@WHERE        ;Get our current address
        RST     28H             ;Call the @WHERE svc
        LD      BC,3+1+1+1+1+2  ;Get the lengths of the instructions after
                                ;the SVC.  This will allow the subroutine
                                ;to return to the correct address.
        ADD     HL,BC           ;Add that offset to where we are
        POP     AF              ;Put stack back
        POP     BC              ;Restore registers
        EX      (SP),HL         ;Put return address on stack and restore HL
        JR      WRITE           ;Jump to the subroutine
                                ;Note that the above was actually a macro
                                ;which performs a relative call.

;       Now we restore the IY register and return to the task scheduler.

        POP     IY              ;Restore IY value
        RET                     ;Return to the task scheduler


;       This routine places characters on the display using the @VDCTL
;       svc instead of @DSP or @DSPLY.  This allows the cursor to
;       remain at its current position when we write to the screen.
;       This routine must be called using the relocatable call macro
;       CALLR.

WRITE:  LD      B,2             ;Put character on the display

TSKLP:  LD      A,(DE)          ;Get a character to display
```

Sample Program F, continued

```
         OR      A              ;Is it time to stop putting this on
                                ;the display?
         RET     Z              ;Yes, return to the caller
         PUSH    HL             ;Save the registers, as the SVC will
         PUSH    DE             ;alter the contents
         PUSH    BC
         LD      C,A            ;Put the character here
         LD      A,@VDCTL       ;Put character on screen at specified position
         RST     28H            ;Call the @VDCTL svc
         POP     BC             ;Restore registers
         POP     DE
         POP     HL
         INC     L              ;Advance display position
         INC     DE             ;Point to next character to display
         JR      TSKLP          ;Loop till date is completely displayed

MODEND:  END     BEGIN          ;End of task and main program
```

Sample Program G

```
;       This program is a sample Extended Command Interpreter.  You
;       may make the ECI as large or small as you require.  You may
;       use allof main memory, or you can restrict yourself to the
;       system overlay area (x'2600' to x'2FFF').
;       To pass a command to the normal system interpreter for
;       processing, use the @CMNDI svc.  TRSDOS executes the command
;       and reloads the ECI.  If you want to have multiple entry
;       points, Bits 2 - 0 in EFLAG$ are in Register A on entry
;       (in Bits 6 - 4),or you may read EFLAG$ yourself.
;       EFLAG$ is totally dedicated to the ECI, and may contain any
;       non-zero value.  If EFLAG$ contains a zero, TRSDOS uses its
;       own interpreter.  Other programs that want to activate an ECI,
;       should set the EFLAG$ to a non-zero value and execute a @EXIT
;       svc.

;       To install an ECI, use the command:
;               COPY filename SYS13/SYS.LSIDOS:d (C=N)
;       If you omit the C=N option, the SYS13 file loses it's "SYS"
;       status and you will receive 'Error 07' messages when you try
;       to use it as a ECI.

;       When SYS1 (the normal command interpreter) has completed it's
;       normal housekeeping and is about to display the "TRSDOS Ready"
;       prompt, it checks EFLAG$.  If EFLAG$ contains a non-zero
;       value, TRSDOS loads and executes the Extended Command
;       Interpreter.
;       To execute this program, type <*><Enter>.

;       This program checks EFLAG$ to see if it is zero.  If so, it
;       sets it to a non-zero value.  This causes this program to be
;       used instead of the normal interpreter when you execute an
;       @EXIT or @ABORT SVC.  (@CMNDI and @CMNDR invoke the TRSDOS
;       interpreter.)  If EFLAG$ is non-zero, the ECI displays a few
;       prompts and the names of all visible /CMD files on logical
;       Drive 0.
;       The operator may then type the name of a program to execute.

;       If you press <Break>,  this program sets EFLAG$ to 0, executes
;       an @EXIT SVC and returns to TRSDOS Ready.

;       By pressing a number, 0 through 7, you can specify the drive
;       that TRSDOS searches.  This program stores this value in
;       EFLAG$.  Each time this program is invoked, it reads the value
;       from EFLAG$ and uses that drive.

;       Note that if a drive is not enabled, not formatted, doesn't
;       exist, or contains no visible /CMD files, this program
;       redisplays the prompt.

        PRINT   SHORT,NOMAC

        PSECT   3000H           ;This program starts at x'3000'

;       Declare the equates for the SVCs used.
;       This is not mandatory, but it makes the program easier to
;       follow.
@EXIT:  EQU     22              ;Exit and return to TRSDOS
@DSPLY: EQU     10              ;Display a string
@FLAGS: EQU     101             ;Locate the system flag area
@DODIR: EQU     34              ;Get the names of filenames
@KEYIN: EQU     9               ;Accept a command and allow editing
@CMNDI: EQU     24              ;Execute a command (using SYS1)

;       On entry, determine if EFLAG$ is set to zero or not.  If it
;       is set to zero, this program is being started by typing
;       PROGRAM<Enter> or <*><Enter>.  In that case, set EFLAG$ to a
;       non-zero value so that in future, TRSDOS uses this interpreter
;       instead of it's own.
```

```
;       If EFLAG$ is non-zero, this initialization has already been
;       done and can be skipped.
BEGIN:  LD      A,@FLAGS        ;Get the starting address of the flag
area
        RST     28H             ;Call the @FLAGS svc
        LD      A,(IY+4)        ;Read the EFLAG$ (ECI flag)
        OR      A               ;Is it set to zero?
        JR      NZ,ECIRUN       ;Run the ECI
        LD      A,8             ;Get a non-zero value.  The value
                                ;needs to be a non-zero value that
                                ;does not set Bits 0, 1 or 2.  The
                                ;default drive # is kept in these bits.
        LD      (IY+4),A        ;Set the EFLAG$ to a non-zero value
        LD      HL,PROMPT       ;Explain how this works
        JR      ECIGO           ;Display message
;       When the system is about to display
;       TRSDOS Ready, it executes this code instead.
ECIRUN: LD      HL,SPROMPT      ;Point at the prompt to use
ECIGO:  LD      A,@DSPLY        ;Display the prompt
        RST     28H             ;Call the @DSPLY svc
;       Display the names of all /CMD files
        LD      A,(IY+4)        ;Get the EFLAG$
        AND     7               ;Delete all but the drive number field
        LD      C,A             ;Store the drive number for the svc
        LD      A,@DODIR        ;Do a directory display
        LD      B,2             ;Display visible, non-system files
        LD      HL,CMDTXT       ;that match "CMD" (stored at CMDTXT)
        RST     28H             ;Call the @DODIR svc
;       Prompt for a filename or a function key.
ASK:    LD      HL,BUFFER       ;Point at text buffer
        LD      B,9             ;Allow up to 8 characters and <Enter>
        LD      C,0             ;Required by the svc
        LD      A,@KEYIN        ;Input text with edit capability
        RST     28H             ;Call the @KEYIN svc
        JR      C,QUIT          ;The carry flag is set when the
                                ;operator presses <BREAK>. Zero the
                                ;EFLAG$ and exit to TRSDOS
        LD      HL,BUFFER       ;Point at the start of the buffer
        LD      A,(HL)          ;Get the character
        CP      0DH             ;Did they type anything?
        JR      Z,ASK           ;No, just repeat the prompt.
                                ;If you want to redisplay the
                                ;directory, change "ASK" to "ECIRUN".
        SUB     '0'             ;Convert value to binary
        CP      7+1             ;Is the character a 0 - 7?
        JR      NC,NAME         ;Must be a filename
;       The operator has typed 1 or more characters that start with
;       a number.  This program assumes that the operator is defining
;       a new drive number and stores this value in EFLAG$ for
;       future use.  TRSDOS does not alter this value.
;       The next time this program is run, EFLAG$ contains the
;       same value and this program knows what drive to scan.
        LD      B,A             ;Save the drive number
        LD      A,(IY+4)        ;Get the EFLAG$
```

```
               AND     8                 ;Delete the old drive number
               OR      B                 ;Insert the new drive number
               LD      (IY+4),A          ;Save that value for future use
               JR      ECIRUN            ;Scan the new drive

        ;      The operator pressed <Break>.  Turn off the ECI and return to
        ;      TRSDOS.
        QUIT:  XOR     A                 ;Get a zero
               LD      (IY+4),A          ;Set EFLAG$ to zero
               LD      HL,EPROMPT        ;Point at the shutdown message
               LD      A,@DSPLY          ;And acknowledge the <Break>
               RST     28H               ;Call the @DSPLY svc
               LD      A,@EXIT           ;Return to TRSDOS Ready
               RST     28H               ;Call the @EXIT svc

        ;      The operator entered what might be a filename or a library
        ;      command.  Pass it to TRSDOS for processing.  If there is an
        ;      error, TRSDOS is responsible for determining what the error is
        ;      and printing a message.
        ;      (HL already points at the start of the buffer.)

        NAME:  LD      A,0DH             ;Look for this character
        FDIV:  CP      (HL)              ;In the command
               JR      Z,FOUND           ;Found the end of the filename
               INC     HL                ;Move character to next byte
               JR      FDIV              ;Find the divider (in this case, a 0DH)

        ;      Found the end of a filename, and add the drive number from
EFLAG$.
        ;      Note that this program may not work properly if the operator
        ;      supplies a drive number as part of the filename.

        FOUND: LD      (HL),':'          ;Add a drive number to the filename
               INC     HL                ;Advance the pointer to the next byte
               LD      A,(IY+4)          ;Get the EFLAG$ value
               AND     7                 ;Delete all but the drive number
               ADD     A,'0'             ;Convert the binary value to ASCII
               LD      (HL),A            ;Add that to the filename
               INC     HL                ;Advance the pointer to the next byte
               LD      (HL),0DH          ;Write a terminator on the end
               LD      HL,BUFFER         ;Point at the text entered
               LD      A,@CMNDI          ;Execute the command, but do not
                                         ;return. Since this program is the
                                         command processor at this time,TRSDOS
                                         ;returns control to the beginning of
                                         ;this module after executing the
                                         ;command.
               RST     28H               ;Call the @CMNDI svc

        ;      Messages and text storage

        PROMPT: DEFM   '[Extended Command Interpreter Is Now Operational]'
               DEFB    0AH
               DEFB    0AH
               DEFM    'Press <BREAK> to use the normal interpreter,
               DEFB    0AH
               DEFM    'type <Number><ENTER> to change the default drive
                       number,'
               DEFB    0AH
               DEFM    'or type the name of the program to run and press
                       <ENTER>'
               DEFB    0DH               ;Terminate the display

        SPROMPT:DEFB   0AH
               DEFM    '[ECI On] <BREAK> to abort, n<ENTER> for new drive or
                       type:'
               DEFM    ' program<ENTER>'
               DEFB    0DH               ;Terminate the message
```

Sample Program G, continued

```
EPROMPT:DEFM    '[Extended Command Interpreter Is Now Disabled]'
        DEFB    ØDH

CMDTXT: DEFM    'CMD'
BUFFER: DEFS    11              ;Allow for filename, drivespec and ØDH

        END     BEGIN           ;"BEGIN" is the starting address
```

9/ Technical Information on TRSDOS Commands and Utilities

TRSDOS commands and utilities are covered extensively in the *Disk System Owner's Manual*. This section presents additional information of a technical nature on several of the commands and utilities.

Changing the Step Rate

The step rate is the rate at which the drive head moves from cylinder to cylinder. You can change the step rate for any drive by using one of the commands described below.

To set the step rate for a particular drive, use the following command:

SYSTEM (DRIVE = *drive*, STEP = *number*)

drive is any drive enabled in the system. *number* can be 0, 1, 2, or 3 and represents one of the following step rates in milliseconds:

0 = 6 milliseconds
1 = 12 milliseconds
2 = 20 milliseconds
3 = 30 milliseconds

Unless it is SYSGENed, the step value you select remains in effect for the specified drive only until the system is re-booted or turned off. If you use the SYSGEN command while the step value is in effect, then this step rate is written to the configuration file (CONFIG/SYS) on the disk in the drive specified by the SYSGEN command.

On a new TRSDOS disk, the step rate is set to 12 milliseconds.

To set the default bootstrap step rate used with the FORMAT utility, use the following command:

SYSTEM (BSTEP = *number*)

number is 0, 1, 2, or 3, which correspond to 6, 12, 20, and 30 milliseconds, respectively.

The value you select for *number* is stored in the system information sector on the disk in Drive 0. (On a new TRSDOS disk, the bootstrap step rate is set to 12 milliseconds.)

If you switch Drive 0 disks or change the logical Drive 0 with the SYSTEM (SYSTEM) command, the default value is taken off the new Drive 0 disk if you format a disk.

You can change the bootstrap step rate for a particular FORMAT operation if you do not want to use the default. Specify the new value for STEP on the FORMAT command line as follows:

FORMAT *:drive* (STEP = *number*)

drive is the drive to be used for the FORMAT. *number* is 0, 1, 2, or 3, which correspond to 6, 12, 20, and 30 milliseconds, respectively.

The step rate is important only if you will be using the disk in Drive 0 to start up the system. Keep in mind that too low a step rate may keep the disk from booting.

Changing the WAIT Value

The WAIT parameter compensates for hardware incompatibility between certain disk drives. The only time you should use it is when *all* tracks above a certain point during a FORMAT operation are shown as locked out when the FORMAT is verified.

The value assigned to WAIT signifies the amount of time between the arrival of the drive head at the location for a read or write, and the actual start of the read or write.

If you want to change the WAIT value, specify the new value on the FORMAT command line as follows:

FORMAT *:drive* (WAIT = *number*)

number is a value between 5000 and 50000. The exact value depends on the particular disk drive you are using. We recommend that you use a value around 25000 at first. Adjust this value higher if tracks are still locked out, or lower until the bottom limit is determined.

Logging in a Diskette

LOG is a utility program that logs in the directory track, number of sides, and density of a diskette. The syntax is:

LOG *:drive*

drive is any drive currently enabled in the system.

The LOG utility provides a way to log in diskette information and update the drive's Drive Code Table (DCT). It performs the same log-in function as the DEVICE library command, except for a single drive rather than all drives. It also provides a way to swap the Drive 0 diskette for a double-sided diskette.

The LOG :0 command prompts you to switch the Drive 0 diskette. You must use this command when switching between double- and single-sided diskettes in Drive 0. Otherwise, it is not needed.

Example

If you want to switch disks in Drive 0, type:

`LOG :0` (ENTER)

The system prompts you with the message:

```
Exchange disks and hit <ENTER>
```

Remove the current disk from Drive 0 and insert the new system disk. When you press (ENTER), information about the new disk is entered to the system.

Printing Graphics Characters

If your printer is capable of directly reproducing the TRS-80 graphics characters, you can use the SYSTEM (GRAPHIC) command. Once you have issued this command, any graphics characters on the screen will be sent to the line printer during a screen print. (Pressing (CTRL)(:) causes the contents of the video display to be printed on the printer.)

Do not use this command unless your printer is capable of directly reproducing the TRS-80 graphics characters.

Changing the Clock Rate

The system normally runs at the fast clock rate of 4 megahertz.

A slow mode of 2 megahertz is available, and may be necessary for real time-dependent programs. (This slow rate is the same as the Model III clock rate.)

To switch to the slow rate, enter the following command:

```
SYSTEM (SLOW)
```

To switch back to the fast rate, enter:

```
SYSTEM (FAST)
```

Appendix A/TRSDOS Error Messages

If the computer displays one of the messages listed in this appendix, an operating system error occurred. Any other error message may refer to an application program error, and you should check your application program manual for an explanation.

When an error message is displayed:

- Try the operation several times.
- Look up operating system errors below and take any recommended actions. (See your application program manual for explanations of application program errors.)
- Try using other diskettes.
- Reset the computer and try the operation again.
- Check all the power connections.
- Check all interconnections.
- Remove all diskettes from drives, turn off the computer, wait 15 seconds, and turn it on again.
- If you try all these remedies and still get an error message, contact a Radio Shack Service Center.

Note: If there is more than one thing wrong, the computer might wait until you correct the first error before displaying the second error message.

This list of error messages is alphabetical, with the binary and hexadecimal error numbers in parentheses. Following it is a quick reference list of the messages arranged in numerical order.

Attempted to read locked/deleted data record (Error 7, X'07')

In a system that supports a "deleted record" data address mark, an attempt was made to read a deleted sector. TRSDOS currently does not use the deleted sector data address mark. Check for an error in your application program.

Attempted to read system data record (Error 6, X'06')

An attempt was made to read a directory cylinder sector without using the directory read routines. Directory cylinder sectors are written with a data address mark that differs from the data sector's data address mark. Check for an error in your application program.

Data record not found during read (Error 5, X'05')

The sector number for the read operation is not on the cylinder being referenced. Either the disk is flawed, you requested an incorrect number, or the cylinder is improperly formatted. Try the operation again. If it fails, use another disk. Reformatting the old disk should lock out the flaw.

Data record not found during write (Error 13, X'0D')

The sector number requested for the write operation cannot be found on the cylinder being referenced. Either the disk is flawed, you requested an incorrect number, or the cylinder is improperly formatted. Try the operation again. If it fails, use another disk.

Device in use (Error 39, X'27')

A request was made to REMOVE a device (delete it from the Device Control Block tables) while it was in use. RESET the device in use before removing it.

Device not available (Error 8, X'08')

A reference was made for a logical device that cannot be found in the Device Control Block. Probably, your device specification was wrong or the device peripheral was not ready. Use the DEVICE command to display all devices available to the system.

Directory full — can't extend file (Error 30, X'1E')

A file has all extent fields of its last directory record in use and must find a spare directory slot but none is available. (See the "Directory Records" section.) Copy the disk's files to a newly formatted diskette to reduce file fragmentation. You may use backup by class or backup reconstruct to reduce fragmentation.

Directory read error (Error 17, X'11')

A disk error occurred during a directory read. The problem may be media, hardware, or program failure. Move the disk to another drive and try the operation again.

Directory write error (Error 18, X'12')

A disk error occurred during a directory write to disk. The directory may no longer be reliable. If the problem recurs, use a different diskette.

Disk space full (Error 27, X'1B')

While a file was being written, all available disk space was used. The disk contains only a partial copy of the file. Write the file to a diskette that has more available space. Then, REMOVE the partial copy to recover disk space.

End of file encountered (Error 28, X'1C')

You tried to read past the end of file pointer. Use the DIR command to check the size of the file. This error also occurs when you use the @PEOF supervisor call to successfully position to the end of a file. Check for an error in your application program.

Extended error (Error 63)

An error has occurred and the extended error code is in the HL register pair.

File access denied (Error 25, X'19')

You specified a password for a file that is not password protected or you specified the wrong password for a file that is password protected.

File already open (Error 41, X'29')

You tried to open a file for UPDATE level or higher, and the file already is open with this access level or higher. This forces a change to READ access protection. Use the RESET library command to close the file.

File not in directory (Error 24, X'18')

The specified filespec cannot be found in the directory. Check the spelling of the filespec.

File not open (Error 38, X'26')

You requested an I/O operation on an unopened file. Open the file before access.

GAT read error (Error 20, X'14')

A disk error occurred during the reading of the Granule Allocation Table. The problem may be media, hardware, or program failure. Move the diskette to another drive and try the operation again.

GAT write error (Error 21, X'15')

A disk error occurred during the writing of the Granule Allocation Table. The GAT may no longer be reliable. If the problem recurs, use a different drive or different diskette.

HIT read error (Error 22, X'16')

A disk error occurred during the reading of the Hash Index Table. The problem may be media, hardware, or program failure. Move the diskette to another drive and try the operation again.

HIT write error (Error 23, X'17')

A disk error occurred during the writing of the Hash Index Table. The HIT may no longer be reliable. If the problem recurs, use a different drive or different diskette.

Illegal access attempted to protected file (Error 37, X'25')

The USER password was given for access to a file, but the requested access required the OWNER password. (See the ATTRIB library command in your *Disk System Owner's Manual.*)

Illegal drive number (Error 32, X'20')

The specified disk drive is not included in your system or is not ready for access (no diskette, non-TRSDOS diskette, drive door open, and so on). See the DEVICE command in your *Disk System Owner's Manual.*)

Illegal file name (Error 19, X'13')

The specified filespec does not meet TRSDOS filespec requirements. See your *Disk System Owner's Manual* for proper filespec syntax.

Illegal logical file number (Error 16, X'10')

A bad Directory Entry Code (DEC) was found in the File Control Block (FCB). This usually indicates that your program has altered the FCB improperly. Check for an error in your application program.

Load file format error (Error 34, X'22')

An attempt was made to load a file that cannot be loaded by the system loader. The file was probably a data file or a BASIC program file.

Lost data during read (Error 3, X'03')

During a sector read, the CPU did not accept a byte from the Floppy Disk Controller (FDC) data register in the time allotted. The byte was lost. This may indicate a hardware problem with the drive. Move the diskette to another drive and try again. If the error recurs, try another diskette.

Lost data during write (Error 11, X'0B')

During a sector write, the CPU did not transfer a byte to the Floppy Disk Controller (FDC) in the time allotted. The byte was lost; it was not transferred to the disk. This may indicate a hardware problem with the drive. Move the diskette to another drive and try again. If the error recurs, try another diskette.

LRL open fault (Error 42, X'2A')

The logical record length specified when the file was opened is different than the LRL used when the file was created. COPY the file to another file that has the specified LRL.

No device space available (Error 33, X'21')

You tried to SET a driver or filter and all of the Device Control Blocks were in use. Use the DEVICE command to see if any non-system devices can be removed to provide more space. This error also occurs on a "global" request to initialize a new file (that is, no drive was specified), if no file can be created.

No directory space available (Error 26, X'1A')

You tried to open a new file and no space was left in the directory. Use a different disk or REMOVE some files that you no longer need.

No error (Error 0)

The @ERROR supervisor call was called without any error condition being detected. A return code of zero indicates no error. Check for an error in your application program.

Parameter error (Error 44,X'2C')

(Under Version 6.2 only) An error occurred while executing a command line or utility because a parameter that does not exist was specified. Check the spelling of the parameter name, value, or abbreviation.

Parity error during header read (Error 1, X'01')

During a sector I/O request, the system could not read the sector header successfully. If this error occurs repeatedly, the problem is probably media or hardware failure. Try the operation again, using a different drive or diskette.

Parity error during header write (Error 9, X'09')

During a sector write, the system could not write the sector header satisfactorily. If this error occurs repeatedly, the problem is probably media or hardware failure. Try the operation again, using a different drive or diskette.

Parity error during read (Error 4, X'04')

An error occurred during a sector read. Its probable cause is media failure or a dirty or faulty disk drive. Try the operation again, using a different drive or diskette.

Parity error during write (Error 12, X'0C')

An error occurred during a sector write operation. Its probable cause is media failure or a dirty or faulty disk drive. Try the operation again, using a different drive or diskette.

Program not found (Error 31, X'1F')

The file cannot be loaded because it is not in the directory. Either the filespec was misspelled or the disk that contains the file was not loaded.

Protected system device (Error 40, X'28')

You cannot REMOVE any of the following devices: *KI, *DO, *PR, *JL, *SI, *SO. If you try, you get this error message.

Record number out of range (Error 29, X'1D')

A request to read a record within a random access file (see the @POSN supervisor call) provided a record number that was beyond the end of the file. Correct the record number or try again using another copy of the file.

Seek error during read (Error 2, X'02')

During a read sector disk I/O request, the cylinder that should contain the sector was not found within the time allotted. (The time is set by the step rate specified in the Drive Code Table.) Either the cylinder is not formatted or it is no longer readable, or the step rate is too low for the hardware to respond. You can set an appropriate step rate using the SYSTEM library command. The problem may also be caused by media or hardware failure. In this case, try the operation again, using a different drive or diskette.

Seek error during write (Error 10, X'0A')

During a sector write, the cylinder that should contain the sector was not found within the time allotted. (The time is set by the step rate specified in the Drive Code Table.) Either the cylinder is not formatted or it is no longer readable, or the step rate is too low for the hardware to respond. You can set an appropriate step rate using the SYSTEM library command. The problem may also be caused by media or hardware failure. In this case, try the operation again, using a different drive or diskette.

— Unknown error code

The @ERROR supervisor call was called with an error number that is not defined. Check for an error in your application program.

Write fault on disk drive (Error 14, X'0E')

An error occurred during a write operation. This probably indicates a hardware problem. Try a different diskette or drive. If the problem continues, contact a Radio Shack Service Center.

Write protected disk (Error 15, X'0F')

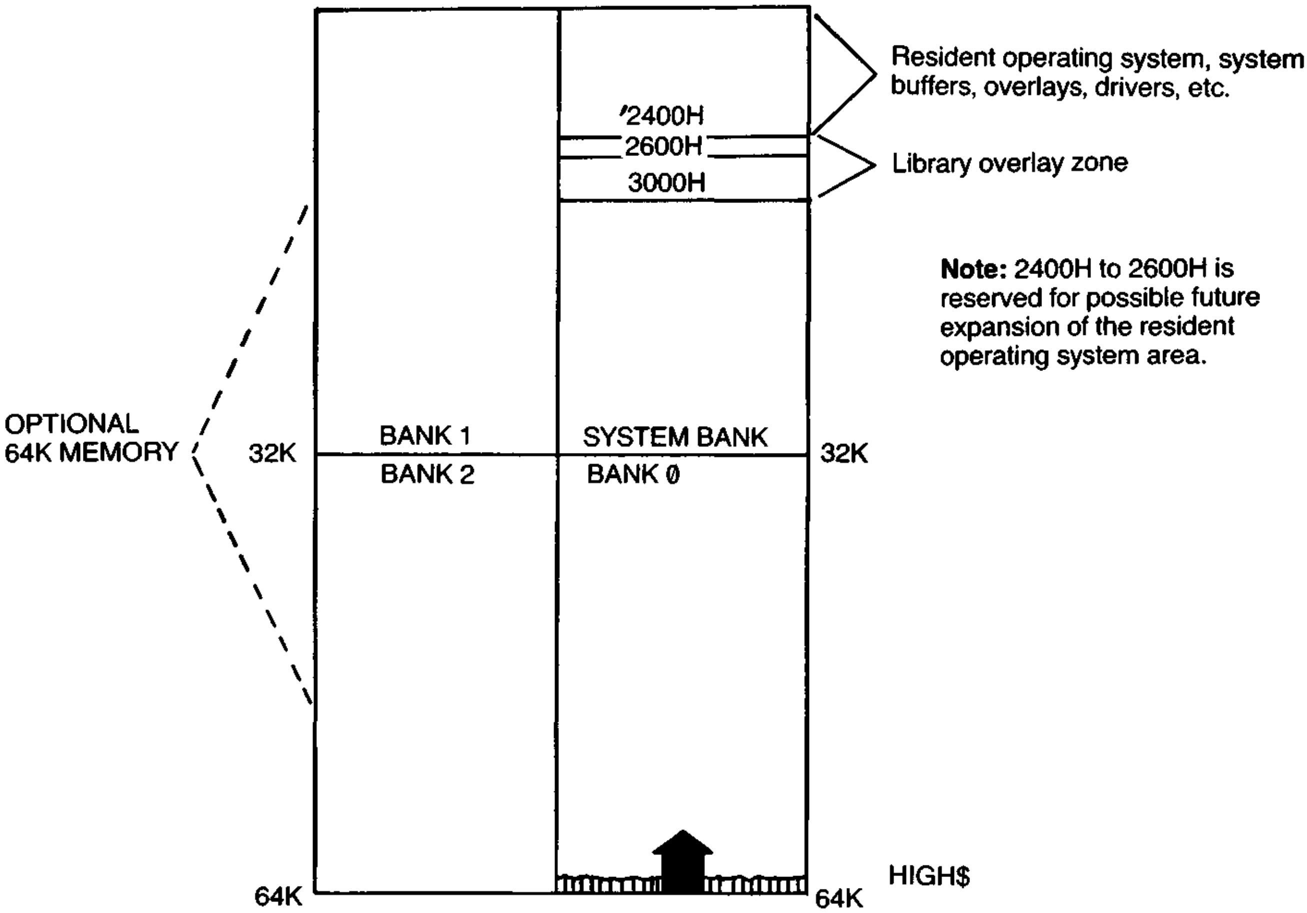
You tried to write to a drive that has a write-protected diskette or is software write-protected. Remove the write-protect tab, if the diskette has one. If it does not, use the DEVICE command to see if the drive is set as write protected. If it is, you can use the SYSTEM library command with the (WP = OFF) parameter to write enable the drive. If the problem recurs, use a different drive or different diskette.

Numerical List of Error Messages

Decimal	Hex	Message
0	X'00'	No Error
1	X'01'	Parity error during header read
2	X'02'	Seek error during read
3	X'03'	Lost data during read
4	X'04'	Parity error during read
5	X'05'	Data record not found during read
6	X'06'	Attempted to read system data record
7	X'07'	Attempted to read locked/deleted data record
8	X'08'	Device not available
9	X'09'	Parity error during header write
10	X'0A'	Seek error during write
11	X'0B'	Lost data during write
12	X'0C'	Parity error during write
13	X'0D'	Data record not found during write
14	X'0E'	Write fault on disk drive
15	X'0F'	Write protected disk
16	X'10'	Illegal logical file number
17	X'11'	Directory read error
18	X'12'	Directory write error
19	X'13'	Illegal file name
20	X'14'	GAT read error
21	X'15'	GAT write error
22	X'16'	HIT read error
23	X'17'	HIT write error
24	X'18'	File not in directory
25	X'19'	File access denied
26	X'1A'	No directory space available
27	X'1B'	Disk space full
28	X'1C'	End of file encountered
29	X'1D'	Record number out of range
30	X'1E'	Directory full—can't extend file
31	X'1F'	Program not found
32	X'20'	Illegal drive number
33	X'21'	No device space available
34	X'22'	Load file format error
37	X'25'	Illegal access attempted to protected file
38	X'26'	File not open
39	X'27'	Device in use
40	X'28'	Protected system device

41	X'29'	File already open
42	X'2A'	LRL open fault
43	X'2B'	SVC parameter error
44	X'2C'	Parameter error
63	X'3F'	Extended error
—		Unknown error code

Appendix B/Memory Map



Note: 2400H to 2600H is reserved for possible future expansion of the resident operating system area.

All software must observe HIGH$.

User software which does not allow TRSDOS library commands to be executed during run time may use memory from 2600H to HIGH$.

User software which allows for library commands during execution must reside in and use memory only between 3000H and HIGH$.

TRSDOS provides all functions and storage through supervisor calls. No address or entry point below 3000H is documented by Radio Shack.

Appendix C/Character Codes

Text, control functions, and graphics are represented in the computer by codes. The character codes range from zero through 255.

Codes one through 31 normally represent certain control functions. For example, code 13 represents a carriage return or "end of line." These same codes also represent special characters. To display the special character that corresponds to a particular code (1-31), precede the code with a code zero.

Codes 32 through 127 represent the text characters — all those letters, numbers, and other characters that are commonly used to represent textual information.

Codes 128 through 191, when output to the video display, represent 64 graphics characters.

Codes 192 through 255, when output to the video display, represent either space compression codes or special characters, as determined by software.

ASCII Character Set

Code Dec.	Code Hex.	ASCII Abbrev.	Keyboard	Video Display
0	00	NUL	CTRL @	Treat next character as displayable; if in the range 1-31, a special character is displayed (see list of special characters later in this Appendix).
1	01	SOH	CTRL A	
2	02	STX	CTRL B	
3	03	ETX	CTRL C	
4	04	EOT	CTRL D	
5	05	ENQ	CTRL E	
6	06	ACK	CTRL F	
7	07	BEL	CTRL G	
8	08	BS	← CTRL H	Backspace and erase
9	09	HT	→ CTRL I	
10	0A	LF	↓ CTRL J	Move cursor to start of next line
11	0B	VT	↑ CTRL K	
12	0C	FF	CTRL L	
13	0D	CR	ENTER CTRL M	Move cursor to start of next line
14	0E	SO	CTRL N	Turn cursor on
15	0F	SI	CTRL O	Turn cursor off
16	10	DLE	CTRL P	Enable reverse video and set high bit routine on*
17	11	DC1	CTRL Q	Set reverse video high bit routine off*
18	12	DC2	CTRL R	
19	13	DC3	CTRL S	
20	14	DC4	CTRL T	
21	15	NAK	CTRL U	Swap space compression/special characters
22	16	SYN	CTRL V	Swap special/alternate characters
23	17	ETB	CTRL W	Set to 40 characters per line
24	18	CAN	SHIFT ← CTRL X	Backspace without erasing
25	19	EM	SHIFT → CTRL Y	Advance cursor
26	1A	SUB	SHIFT ↓ CTRL Z	Move cursor down
27	1B	ESC	SHIFT ↑ CTRL ,	Move cursor up
28	1C	FS	CTRL /	Move cursor to upper left corner. Disable reverse video and set high bit routine off.* Set to 80 characters per line.
29	1D	GS	CTRL ENTER CTRL .	Erase line and start over
30	1E	RS	CTRL ;	Erase to end of line

*When the high bit routine is on, characters 128 through 191 are displayed as standard ASCII characters in reverse video.

Code Dec.	Hex.	ASCII Abbrev.	Keyboard	Video Display
31	1F	VS	(SHIFT)(CLEAR)	Erase to end of display
32	20	SPA	(SPACEBAR)	(blank)
33	21		(!)	!
34	22		(")	"
35	23		(#)	#
36	24		($)	$
37	25		(%)	%
38	26		(&)	&
39	27		(')	'
40	28		(()	(
41	29		())	)
42	2A		(*)	*
43	2B		(+)	+
44	2C		(,)	,
45	2D		(-)	-
46	2E		(.)	.
47	2F		(/)	/
48	30		(0)	0
49	31		(1)	1
50	32		(2)	2
51	33		(3)	3
52	34		(4)	4
53	35		(5)	5
54	36		(6)	6
55	37		(7)	7
56	38		(8)	8
57	39		(9)	9
58	3A		(:)	:
59	3B		(;)	;
60	3C		(<)	<
61	3D		(=)	=
62	3E		(>)	>
63	3F		(?)	?
64	40		(@)	@
65	41		(SHIFT)(A)	A
66	42		(SHIFT)(B)	B
67	43		(SHIFT)(C)	C
68	44		(SHIFT)(D)	D
69	45		(SHIFT)(E)	E
70	46		(SHIFT)(F)	F
71	47		(SHIFT)(G)	G
72	48		(SHIFT)(H)	H
73	49		(SHIFT)(I)	I
74	4A		(SHIFT)(J)	J
75	4B		(SHIFT)(K)	K
76	4C		(SHIFT)(L)	L
77	4D		(SHIFT)(M)	M
78	4E		(SHIFT)(N)	N
79	4F		(SHIFT)(O)	O
80	50		(SHIFT)(P)	P
81	51		(SHIFT)(Q)	Q
82	52		(SHIFT)(R)	R
83	53		(SHIFT)(S)	S
84	54		(SHIFT)(T)	T
85	55		(SHIFT)(U)	U
86	56		(SHIFT)(V)	V
87	57		(SHIFT)(W)	W
88	58		(SHIFT)(X)	X
89	59		(SHIFT)(Y)	Y

Code Dec.	Hex.	ASCII Abbrev.	Keyboard	Video Display
90	5A		SHIFT Z	Z
91	5B		CLEAR ,	[
92	5C		CLEAR /	\
93	5D		CLEAR .	]
94	5E		CLEAR ;	^
95	5F		CLEAR ENTER	—
96	60		SHIFT @	
97	61		A	a
98	62		B	b
99	63		C	c
100	64		D	d
101	65		E	e
102	66		F	f
103	67		G	g
104	68		H	h
105	69		I	i
106	6A		J	j
107	6B		K	k
108	6C		L	l
109	6D		M	m
110	6E		N	n
111	6F		O	o
112	70		P	p
113	71		Q	q
114	72		R	r
115	73		S	s
116	74		T	t
117	75		U	u
118	76		V	v
119	77		W	w
120	78		X	x
121	79		Y	y
122	7A		Z	z
123	7B		CLEAR SHIFT ,	{
124	7C		CLEAR SHIFT /	|
125	7D		CLEAR SHIFT .	}
126	7E		CLEAR SHIFT ;	~
127	7F	DEL	CLEAR SHIFT ENTER	±

Extended (non-ASCII) Character Set

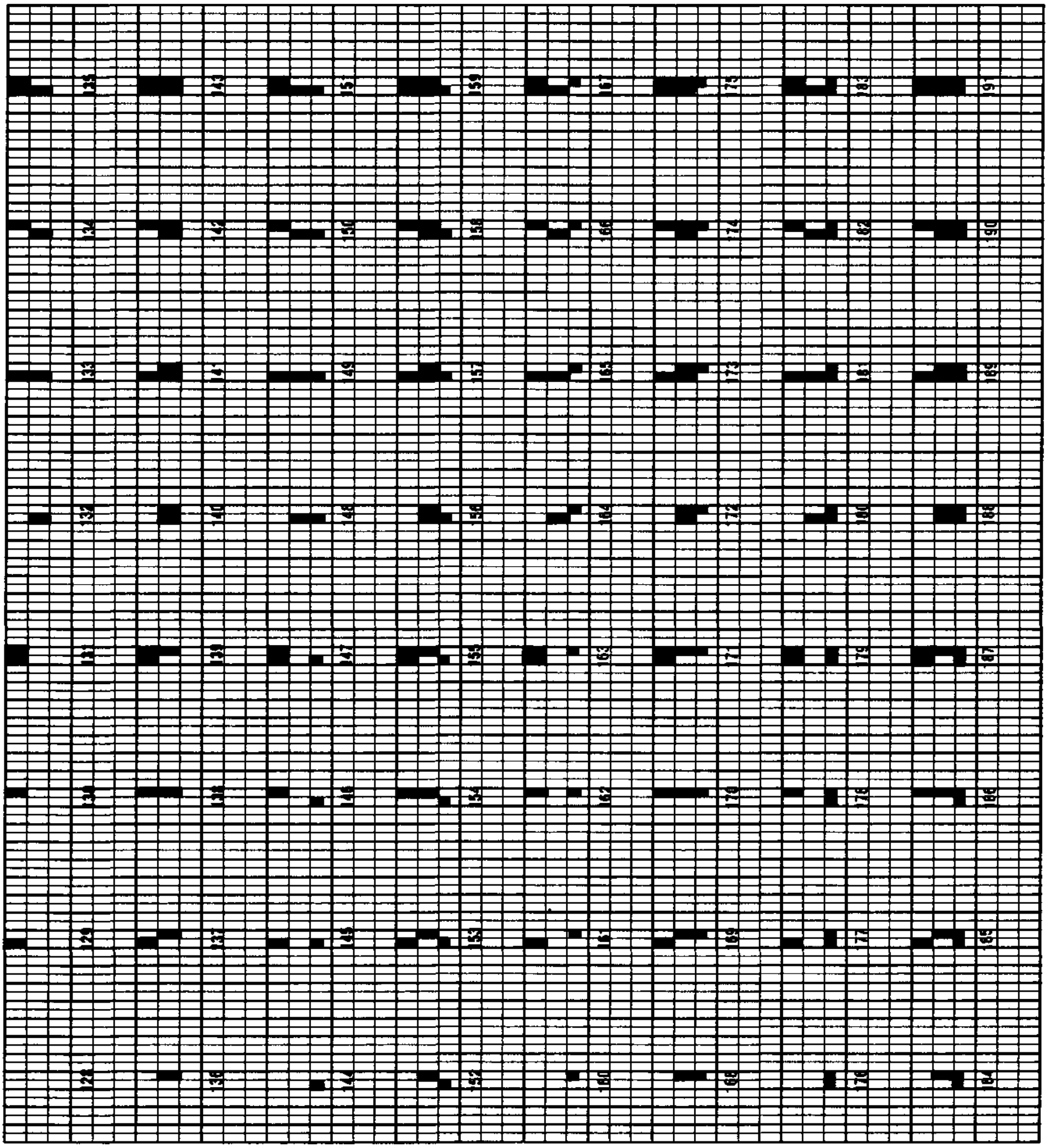
Code Dec.	Hex.	Keyboard	Video Display
128	80	BREAK	See graphics character table in this Appendix.
129	81	F1 CLEAR CTRL A	
130	82	F2 CLEAR CTRL B	
131	83	F3 CLEAR CTRL C	
132	84	CLEAR CTRL D	
133	85	CLEAR CTRL E	
134	86	CLEAR CTRL F	
135	87	CLEAR CTRL G	
136	88	CLEAR CTRL H	
137	89	CLEAR CTRL I	
138	8A	CLEAR CTRL J	
139	8B	CLEAR CTRL K	
140	8C	CLEAR CTRL L	
141	8D	CLEAR CTRL M	
142	8E	CLEAR CTRL N	
143	8F	CLEAR CTRL O	
144	90	CLEAR CTRL P	
145	91	SHIFT F1 CLEAR CTRL Q	
146	92	SHIFT F2 CLEAR CTRL R	
147	93	SHIFT F3 CLEAR CTRL S	
148	94	CLEAR CTRL T	
149	95	CLEAR CTRL U	
150	96	CLEAR CTRL V	
151	97	CLEAR CTRL W	
152	98	CLEAR CTRL X	
153	99	CLEAR CTRL Y	
154	9A	CLEAR CTRL Z	
155	9B	CLEAR SHIFT ↑	
156	9C		
157	9D		
158	9E		
159	9F		
160	A0	CLEAR SPACE	
161	A1	CLEAR SHIFT 1	
162	A2	CLEAR SHIFT 2	
163	A3	CLEAR SHIFT 3	
164	A4	CLEAR SHIFT 4	
165	A5	CLEAR SHIFT 5	
166	A6	CLEAR SHIFT 6	
167	A7	CLEAR SHIFT 7	
168	A8	CLEAR SHIFT 8	
169	A9	CLEAR SHIFT 9	
170	AA	CLEAR SHIFT :	
171	AB		
172	AC		
173	AD	CLEAR -	
174	AE		
175	AF		
176	B0	CLEAR 0	
177	B1	CLEAR 1	
178	B2	CLEAR 2	

Code Dec.	Hex.	Keyboard	Video Display
179	B3	CLEAR 3	See graphics character table in this Appendix.
180	B4	CLEAR 4	
181	B5	CLEAR 5	
182	B6	CLEAR 6	
183	B7	CLEAR 7	
184	B8	CLEAR 8	
185	B9	CLEAR 9	
186	BA	CLEAR :	
187	BB		
188	BC		
189	BD	CLEAR SHIFT -	
190	BE		
191	BF		
192	C0	CLEAR @*	See list of special characters in this Appendix.
193	C1	CLEAR A**	
194	C2	CLEAR B**	
195	C3	CLEAR C**	
196	C4	CLEAR D**	
197	C5	CLEAR E**	
198	C6	CLEAR F**	
199	C7	CLEAR G**	
200	C8	CLEAR H**	
201	C9	CLEAR I**	
202	CA	CLEAR J**	
203	CB	CLEAR K**	
204	CC	CLEAR L**	
205	CD	CLEAR M**	
206	CE	CLEAR N**	
207	CF	CLEAR O**	
208	D0	CLEAR P**	
209	D1	CLEAR Q**	
210	D2	CLEAR R**	
211	D3	CLEAR S**	
212	D4	CLEAR T**	
213	D5	CLEAR U**	
214	D6	CLEAR V**	
215	D7	CLEAR W**	
216	D8	CLEAR X**	
217	D9	CLEAR Y**	
218	DA	CLEAR Z**	
219	DB		
220	DC		
221	DD		
222	DE		
223	DF		
224	E0	CLEAR SHIFT @	
225	E1	CLEAR SHIFT A	
226	E2	CLEAR SHIFT B	
227	E3	CLEAR SHIFT C	
228	E4	CLEAR SHIFT D	
229	E5	CLEAR SHIFT E	
230	E6	CLEAR SHIFT F	
231	E7	CLEAR SHIFT G	
232	E8	CLEAR SHIFT H	
233	E9	CLEAR SHIFT I	
234	EA	CLEAR SHIFT J	

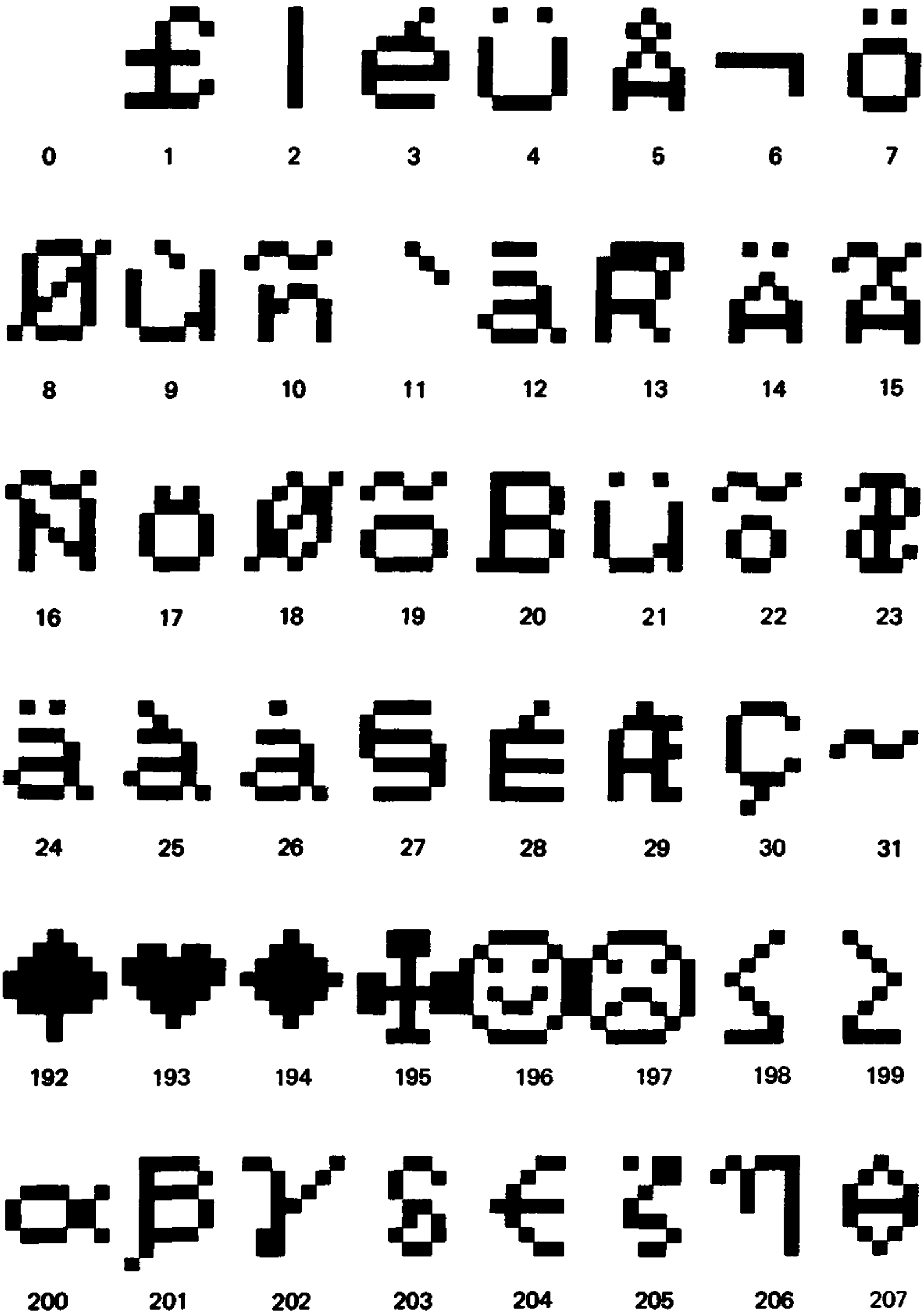
*Empties the type-ahead buffer.
**Used by Keystroke Multiply, if KSM is active.

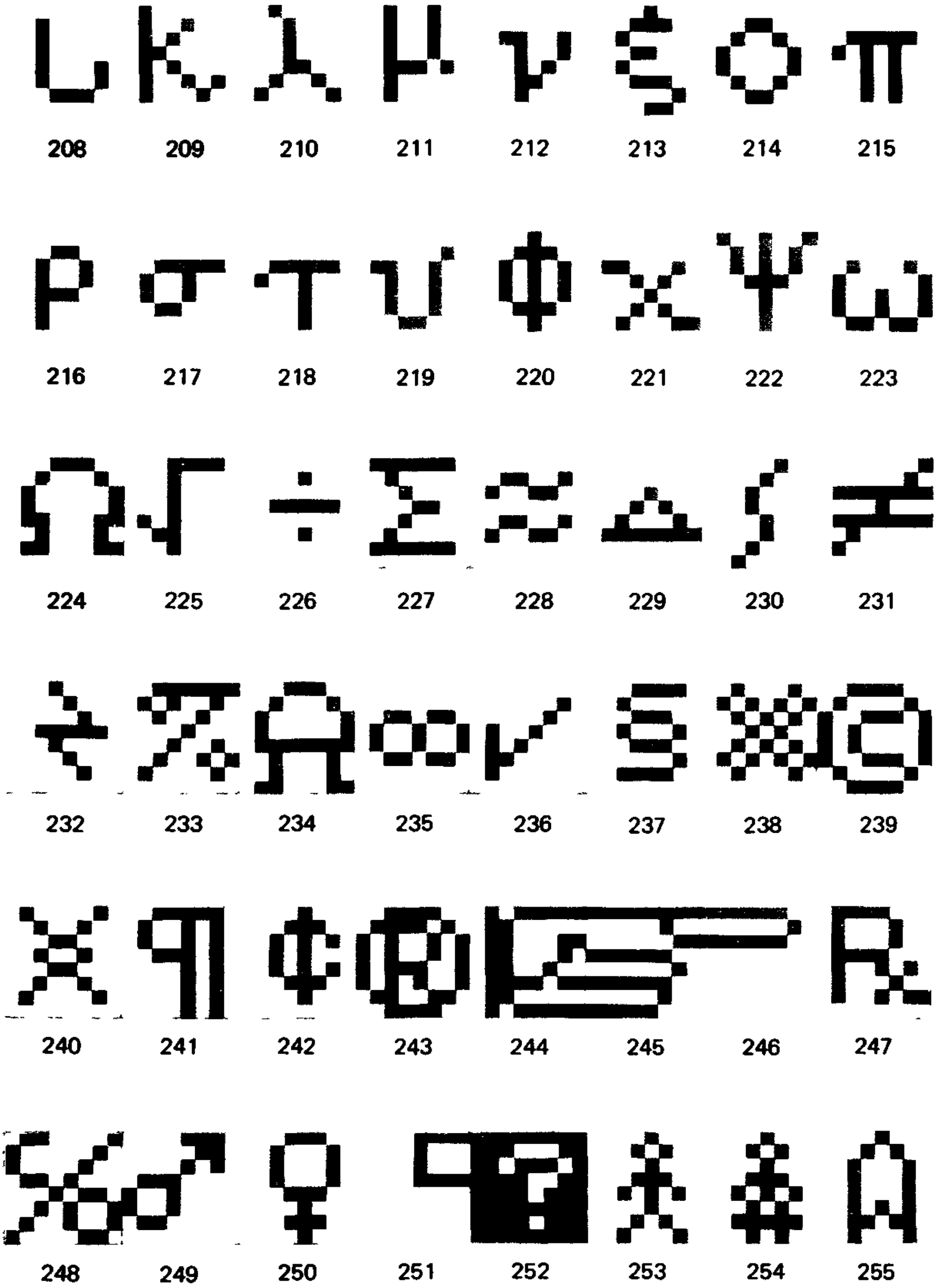
Code Dec.	Code Hex.	Keyboard	Video Display
235	EB	CLEAR SHIFT K	See list of special characters in this Appendix.
236	EC	CLEAR SHIFT L	
237	ED	CLEAR SHIFT M	
238	EE	CLEAR SHIFT N	
239	EF	CLEAR SHIFT O	
240	F0	CLEAR SHIFT P	
241	F1	CLEAR SHIFT Q	
242	F2	CLEAR SHIFT R	
243	F3	CLEAR SHIFT S	
244	F4	CLEAR SHIFT T	
245	F5	CLEAR SHIFT U	
246	F6	CLEAR SHIFT V	
247	F7	CLEAR SHIFT W	
248	F8	CLEAR SHIFT X	
249	F9	CLEAR SHIFT Y	
250	FA	CLEAR SHIFT Z	
253	FD		
254	FE		
255	FF		

Graphics Characters (Codes 128-191)



Special Characters (0-31, 192-255)




208 209 210 211 212 213 214 215
216 217 218 219 220 221 222 223
224 225 226 227 228 229 230 231
232 233 234 235 236 237 238 239
240 241 242 243 244 245 246 247
248 249 250 251 252 253 254 255

Appendix D/Keyboard Code Map

The keyboard code map shows the code that TRSDOS returns for each key, in each of the modes: control, shift, unshift, clear and control, clear and shift, clear and unshift.

For example, pressing (CLEAR), (SHIFT), and (1) at the same time returns the code X'A1'.

A program executing under TRSDOS — for example, BASIC — may translate some of these codes into other values. Consult the program's documentation for details.

(BREAK) Key Handling

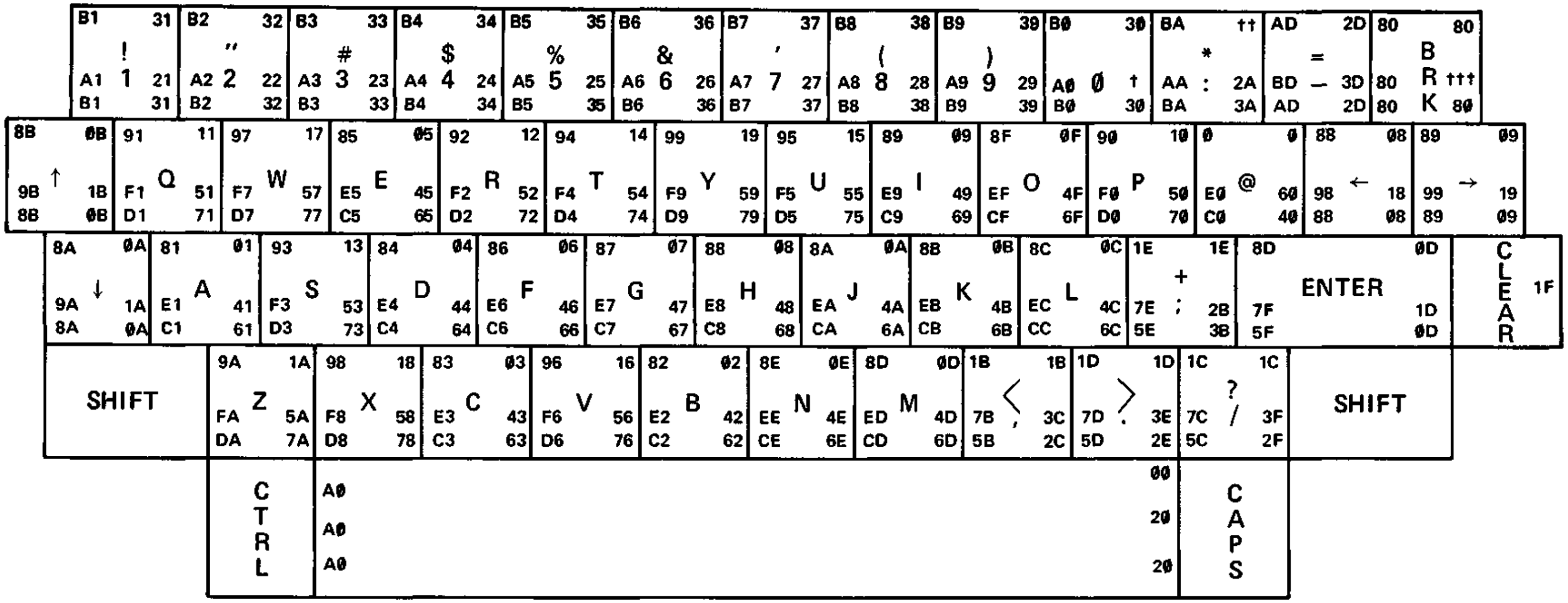
The (BREAK) key (X'80') is handled in different ways, depending on the settings of three system functions. The table below shows what happens for each combination of settings.

Break Enabled	Break Vector Set	Type-Ahead Enabled	
Y	N	Y	If characters are in the type-ahead buffer, then the buffer is emptied.* If the type-ahead buffer is empty, then a BREAK character (X'80') is placed in the buffer.*
Y	N	N	A BREAK character (X'80') is placed in the buffer.
Y	Y	Y	The type-ahead buffer is emptied of its contents (if any), and control is transferred to the address in the BREAK vector (see @BREAK SVC).*
Y	Y	N	Control is transferred to the address in the BREAK vector (see @BREAK SVC).
N	X	X	No action is taken and characters in the type-ahead buffer are not affected.

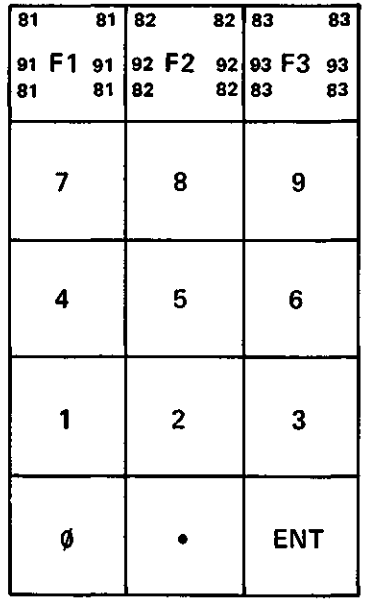
*Because the (BREAK) key is checked for more frequently than other keys on the keyboard, it is possible for (BREAK) to be pressed after another key on the keyboard and yet be detected first.

Y means that the function is on or enabled
N means that the function is off or disabled
X means that the state of the function has no effect

Break is enabled with the SYSTEM (BREAK = ON) command (this is the default condition).
The break vector is set using the @BREAK SVC (normally off).
Type-ahead is enabled using the SYSTEM (TYPE = ON) command (this is the default condition).

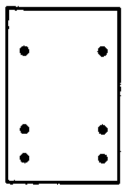


The keys may be positioned differently on your keyboard. However, they produce the same codes.



Codes for these keys are the same as for the main keyboard.

LEGEND:



Clear and Control — Control

Clear and Left Shift — Shift

Clear and Unshift — Unshift

Note: Pressing CONTROL, SHIFT, and @ at the same time generates an EOF (end of file) — — X'1C' with NZ return flag.

Whenever pressing CLEAR, SHIFT, and another key at the same time, be sure to use the left SHIFT key — not the right SHIFT key.

† Pressing SHIFT and Ø at the same time (or CAPS alone) turns the CAPS mode on or off.

†† Pressing CONTROL and : at the same time causes a screen print.

††† Pressing SHIFT and BREAK at the same time reselects the last drive.

Appendix E/Programmable SVCs

(Under Version 6.2 only)

SVC numbers 124 through 127 are reserved for programmer installable SVCs. To install an SVC the programmer must write the routine to execute when the SVC is called.

The routine should be written as high memory module if it is to be available at all times. If you execute a SYSGEN command when a programmable SVC is defined, the address of the routine is saved in the SYSGEN file and restored each time the system is configured. If the routine is a high memory module, the routine is saved and restored as well. This makes the SVC always available. For more information on high memory modules, see Memory Header and Sample Program F.

To install an SVC, the program must access the SVC table. The SVC table contains 128 two-byte positions, a two-byte position for each usable SVC. Each position in the table contains the address of the routine to execute when the SVC is called.

To access the SVC table, execute the @FLAGS SVC (SVC 101). IY + 26 contains the MSB of the SVC table start address. The LSB of the SVC table address is always 0 because the SVC table always begins on a page boundary.

Store the address of the routine to be executed at the *SVC number times 2* byte in the table. For example, if you are installing SVC 126, store the address of the routine at byte 252 in the table. Addresses are stored in LSB-MSB format.

When the SVC is executed, control is transferred to the address in the table. On entry to your SVC, Register A contains the same value as Register C. All other registers retain the values they had when the RST 28 SVC instruction was executed.

To exit the SVC, execute a RET instruction. The program should save and restore any registers used by the SVC.

Initially, SVCs 124 through 127 display an error message when they are executed. When installing an SVC you should save the original address at that location in the table and restore it when you remove the SVC.

These program lines insert a new SVC into the system SVC table, save the previous value of the table, and reinsert that value before execution ends. You could check the existing value to see if the address is above X'2600'. If it is, the SVC is already assigned and should not be used at this time.

This code inserts SVC 126, called MYSVC:

```
LD      A,@FLAGS        ;Locate start of SVC table
RST     28H             ;Execute @FLAGS SVC
LD      H,(IY+26)       ;Get MSB of address
LD      L,126*2         ;Want to use SVC 126
LD      (OSVC126A),HL   ;Save address of SVC entry
LD      E,(HL)          ;Get current SVC address
INC     HL
LD      D,(HL)
LD      (OSVC126V),DE   ;Save the old value
DEC     HL
LD      DE,MYSVC        ;Get address of routine for
                        ;SVC 126
LD      (HL),E          ;Insert new SVC address into
                        ;table
INC     HL
```

```
LD          (HL),D
.
. Code that uses MYSVC (SVC 126)
.
.
```

This code removes SVC 126:

```
LD          HL,(OSVC126A)     ;Get address of SVC entry
LD          DE,(OSVC126V)     ;Get original value
LD          (HL),E            ;Insert original SVC address
INC         HL
LD          (HL),D
```

Appendix F/Using SYS13/SYS

(Under Version 6.2 only)

With TRSDOS Version 6.2, you can create an Extended Command Interpreter (ECI) or an Immediate Execution Program (IEP). TRSDOS can store either an ECI or IEP in the SYS13 file. Both programs cannot be present at the same time.

At the TRSDOS Ready prompt when you type (*) (ENTER), TRSDOS executes the program stored in SYS13/SYS. Because TRSDOS recognizes the program as a system file, TRSDOS includes the file when creating backups and loads the program faster.

If you want to write additional commands for TRSDOS, you can write an interpreter to execute these commands. Your ECI can also execute TRSDOS commands by using the @CMNDI SVC to pass a command to the TRSDOS interpreter.

If EFLAG$ contains a non-zero value, TRSDOS executes the program in SYS13/SYS. If EFLAG$ contains a zero, TRSDOS uses its own command interpreter.

Sample Program G is an example of an ECI. It is important to note that your ECI must be executable by pressing (*) (ENTER) at the TRSDOS Ready prompt.

An ECI can use all of memory or you can restrict it to use the system overlay area (X'2600' to X'2FFF').

To implement an IEP or ECI, use the following syntax:

COPY *filespec* SYS13/SYS.LSIDOS:*drive* (C = N) (ENTER)

filespec can be any executable (/CMD) program file. *drive* specifies the destination drive. The destination drive must contain an original SYS13/SYS file.

Example

COPY SCRIPSIT/CMD:1 SYS13/SYS.LDI:0 (C = N)

TRSDOS copies SCRIPSIT/CMD from Drive 1 to SYS13/SYS in Drive 0. At the TRSDOS Ready prompt, when you press (*) (ENTER), TRSDOS executes SCRIPSIT.

Index

Index

Index

Index

Index

Subject	Page	Subject	Page

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG AVENUE MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL 5140 NANINNE (NAMUR)	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

S-L/3-85

Printed in U.S.A.