CHROMATICS

CGC 7900 COLOR GRAPHICS COMPUTER SYSTEM

PRELIMINARY DOS MANUAL
(with Assembler and Text Editor)

July, 1981

2558 Mountain Industrial Boulevard
Tucker, Georgia 30084

Phone (404) 493-7000
TWX 810-766-8099

This document is an advance release, provided for informational purposes only. The specifications contained herein are subject to revsion prior to shipment of the product.

PUBLICATION COMMENT FORM

ease use this sheet to give us feedback on the enclosed cumentation. Your comments on errors or omissions, suggested anges, format of presentation, etc., will be helpful in eparing future manuals. Wherever possible, refer to specific ge numbers in your comments below. After completing this rm, mail to:

Chromatics, Inc.
2558 Mountain Industrial Boulevard
Tucker, GA 30084

Attn: R & D Department

Name:_____ Date:_____

Company Name: _____

Name/Model Number of Product:_____

Title of Publication: _____

Date of Publication Release: _____ Preliminary?_____

Page No.	Comments

CONVENTIONS USED IN THIS DOCUMENT

1. Any keys which have labeled caps will be called by their full names, capitalized and underlined. For example, the carriage return key will be denoted by

RETURN

2. The modifier keys, CTRL, SHIFT, M1, and M2, must be held down while striking the key they are to modify. Note that these four keys do not generate any characters on their own, but simply modify the character which is struck simultaneously. This process of holding down a modifier key while striking another key will be denoted by the modifier AND the key being underlined together. For example,

CTRL F

would indicate that the CTRL key should be held down while striking the F key. If two or more modifiers are needed simultaneously, they will all be underlined together:

CTRL SHIFT T

would mean that BOTH modifiers, SHIFT and CTRL, should be held down while striking the T key.

3. Variable parameters will be enclosed in angle brackets, < >. Any items enclosed in these brackets will be explained in full in the text which immediately follows.

4. Optional parameters will be enclosed in square brackets [ ]. Any items which may be repeated will be followed by an ellipsis (three dots).

Example of (3) and (4):

<X>, [<Y1>,<Y2>,...]

The parameter <X> is required. The parameters <Y1>, <Y2>, and so on, are optional. Any number of these may be included. All three types of parameters would be explained immediately beneath the example which contained them.

5. Zeros will be slashed (Ø), alphabetic O will not be slashed.

TABLE OF CONTENTS

SECTION 1 - THE DISK OPERATING SYSTEM

INTRODUCTION

This is the manual for the Chromatics CGC 7900 Disk Operating System (DOS), an optional feature of the 7900 series of color graphic computers. The Disk Operating System uses two double-density flexible disk drives for program and data storage. A fixed disk drive with 10 megabytes of storage is also available. The DOS option consists of these parts:

Disk drives

PROMs (firmware)

A diskette with system programs

This manual

The disk drives and PROMs are factory-installed, and should require no attention by you (except that the fixed disk may require special unpacking; instructions for this are attached to your unit if applicable).

The diskette contains programs which provide an interface between the disk drives and your programs or data. Routines are provided to save data onto a disk, to retrieve data from a disk, and to manipulate the contents of the disks. This diskette should be handled carefully as you are learning to use the DOS. You should make a copy of this diskette at your earliest opportunity. Instructions for copying a diskette are included in this manual (see the FORMAT and COPY commands).

This manual begins by describing the commands and utility routines available in DOS. In later sections, the Text Editor and MC68000 Resident Assembler are discussed. These two programs allow you to create text files and assembly language programs on the disk, and to generate executable binary machine code for the 68000 processor.

Detailed descriptions of the CGC 7900 special features, such as the color graphics plotting capability, are not provided here. Please refer to your User's Manual for information on other aspects of the CGC 7900.

DISKETTES

DOS stores information on the surface of disks, which are coated with a magnetic material. Flexible diskettes, or "floppy disks," are a very reliable and convenient way to store data. A flexible disk will perform well for many hours of use, if a few simple precautions are observed:

HANDLING - DO NOT touch the exposed surface of the diskette, which is visible through a slot on either side of the diskette. DO NOT attempt to remove the circular diskette from its square, dark envelope. Handle the diskette carefully, and do not fold it.

LABELING - A diskette is provided with adhesive labels which should be used to note the contents of the diskette. Write on these labels BEFORE attaching the label to the diskette. If you must write on a label after it has been attached to the diskette, use a felt-tip pen and press gently. A ball-point pen will crease the disk and may cause permanent damage.

INSERTING - To insert a diskette into a drive, first remove the diskette from its paper sleeve. Hold the diskette gently, with the label UP, and the arrow on the label aiming toward the drive. Open the drive door by pressing the rectangular button until the door snaps open. GENTLY slide the diskette into the drive until it is completely inside the door (it may seat with a soft "click"). Press the door shut. To remove the diskette, press the rectangular button again.

STORAGE - When a diskette is not in use, it should be removed from the drive and stored in its paper sleeve. Store the diskette away from dust, away from extremely high or low temperatures, away from moisture, and AWAY FROM MAGNETIC FIELDS. Protect the diskette from magnets, motors, transformers, or anything else which could create magnetism.

OPERATION - When a disk drive is in use, the red light in the drive door will illuminate. It is extremely important that nothing interfere with the disk while this light is on. While a disk is in use, removing the disk, pressing RESET on the keyboard, or turning the power off, may damage the data on a disk.

The fixed disk is a sealed unit, located in the base of the 7900 chassis. It is not subject to many of the restrictions above, since it is hidden away from normal view. But the warning about interrupting a disk operation in progress is still valid: if you have any reason to believe the system is accessing your fixed disk, DO NOT press RESET or otherwise interrupt the process. If you give the system a command to access the fixed disk, be sure the command has been completed before turning the system off or pressing RESET.

WRITE-PROTECTION - A flexible diskette may be protected from accidental destruction by uncovering its write-protect notch. Some diskettes are shipped with the notch covered, and some have it uncovered when you receive them. In either case, the notch must be covered or DOS will not be able to write on the disk. The write-protect notch is a small (1/4") hole on the front edge of the disk.

ENTRY INTO DOS

The Disk Operating System is entered by pressing the labeled key,

DOS

The DOS log-on message should immediately appear on the screen. If this does not happen, it can be because the state of the system is not what DOS expects to find (for example, if the screen is not connected as the proper output device). You can optionally enter the Disk Operating System by striking three keys,

RESET CTRL BOOT DOS

This sequence initializes the entire system and will always cause entry into DOS.

The DOS log-on message should now appear:

```
CGC Disk Operating System --- Version 1.4
Copyright (c) 1981 by CHROMATICS, INC.

ENTER USER PASSWORD =
```

DOS will print its version number. This number should be noted in any communications to Chromatics concerning the DOS.

The DOS log-on message will request your User Password. At this point, you may enter a 2-character password and press the RETURN key, or you may simply press RETURN. If you do not enter a password, you will only have access to Public files which are not assigned a password. If you do enter a password, you will have access to all Public files, as well as any files whose password matches yours.

Legal characters for a User Password are: digits 0-9, upper and lower case alphabetic characters, and these special characters:

```
[ \ ] ^ _ ` { | } ~
```

Entering any other characters may cause the system to ignore your password and assign you to Public files.

NOTE: The User Password system in DOS is not designed to offer a high level of protection. Its main purpose is to help organize files into groups, so that a user will see only the files he must work with. This is especially important in the case of the hard disk, where several hundred files may exist in the directory.

After completing the log-on procedure, DOS prints a green asterisk (*) as its prompt character. The asterisk means that DOS is ready to accept a command.

DOS COMMAND LINE

When you are entering commands to DOS, all of the text editing functions labeled on the cursor keypad may be used to edit your input line. The left and right arrow keys move the cursor around on the input line. the HOME key moves the cursor to the beginning of the input line. The functions labeled in blue are accessed by holding the CTRL modifier and pressing the indicated key: these functions are Insert Character, Delete Character, Clear Line, and Clear EOL. Pressing RECALL brings back a copy of previous lines. RECALL and SHIFT RECALL may be used to retrieve any line from the "Recall Buffer." Once recalled, a previously entered line may be edited with the other functions.

Regardless of where the cursor is on the input line, ALL characters visible on the input line are accepted when the RETURN key is pressed. DOS does nothing with your commands until you press RETURN. If you press DELETE instead of RETURN, DOS ignores the line you typed.

(All of these functions are a part of the Inline Editor, used for DOS, the Monitor, and other 7900 programs. The Inline Editor is discussed in more detail in the 7900 User's Manual, and in Section Two of this manual.)

DOS commands are described in detail in this manual. Most are simple words or abbreviations, such as

```
DIR   (followed by RETURN)
```

which lists the directory of a disk (the names of the files on that disk).

You may enter several DOS commands on the same line, separating them by a colon (:). For example, the following command would list the disk directories from drives 1 and 2:

```
DIR/1:DIR/2
```

(No space should be typed on either side of the colon.) You may type as many commands as will fit on a single line of the screen. If any command causes an error, DOS will not process the rest of the commands on that line.

DISK FILE NAMES

DOS is a file oriented system. When you type a command to DOS, you are actually entering the name of a disk file. DOS looks for the file, and if the file is found (and is executable), it is loaded into memory and executed.

A file name may have several parts:

The primary name, which may be one to eight alphanumeric characters, and may contain the special characters

[\ ] ^ _ ` { | } ~

The secondary name, which is always three alphanumeric characters. The secondary name is separated from the primary name by a period (decimal point). The secondary name is optional. If it is omitted, DOS will usually append a secondary name to the file name (depending on what the file name is to be used for).

A password, which must match the password assigned to the file. The password is separated from the name by a dollar sign ($). If a password is entered, it must precede the drive number. The password is optional. If it is omitted, it is assumed to be the password under which you logged onto DOS.

A drive number, identifying the disk drive on which the file resides. The drive number is separated from the name by a slash (/). The drive number is optional. If it is omitted, it will be assumed this file resides on the same drive which responded to the last DOS command (the drive from which the last command was loaded).

The four parts of a file name (primary name, secondary name, password, drive number) must occur in the order listed. If any of the optional parts are omitted, the remaining parts must occur in the required order.

Examples of legal file names:

FILEname

STORY.SRC

PROGRAM/2

USRT29NE$PW

LISTING.OBJ/1

HardLuck.BUF$ED/3

Examples of improper file names:

THISISTOOLONG	Too many characters in primary name - only the first eight characters would be recognized, so this is equivalent to THISISTO. It would, however, still be accepted.
oops/A	Illegal drive number.
NOWAY!.SYS	Illegal character in primary name.
BADone.GO	Secondary name too short.
Wrong/1.SRC	Incorrect order.

DISK DRIVE NUMBERS

The CGC 7900 supports up to three disk drives: two flexible disks and a hard disk. Many disk commands require specifying the drive number of the disk to which the command refers. The following numbers apply:

Drive 1: the left-hand flexible disk

Drive 2: the right-hand flexible disk

Drive 3: the hard disk

You may always specify the drive number if you wish. Anytime you do not specify a drive number, DOS assumes you are still using the same drive you used in a previous command.

If a drive number of 0 (zero) is entered, it implies that DOS should search all drives to locate the requested file. The search begins with drive 1. (If a new file is being created, the drive number must be implicitly or explicitly specified, so a drive number of zero is not allowed.)

When you enter DOS, immediately after pressing the DOS key, the system does not know which drive you want to use. The FIRST command you enter to DOS acts as if you had specified a drive number of zero, so DOS will search all drives in your system in an attempt to execute the command. If this search succeeds, DOS now knows which drive you want to use, and it will stay with that drive until you specify a different drive number.

On the other hand, if your first command to DOS fails (as it would if you misspelled a command), DOS will display an error message. DOS will now default back to drive 1. The next command you enter will assume that drive 1 is in use. Therefore, if an error occurs in your FIRST command to DOS, let your SECOND command specify a drive number unless you want to use drive 1.

The feature of "remembering" the current drive applies only to commands. DOS only remembers where the last COMMAND came from, not the last filename. If you type:

```
KILL/2 filename/3
```

The KILL command is coming from drive 2, so DOS remembers drive 2 and will search it for the next command (unless you specify a different drive number).

SECONDARY FILE NAMES

The following secondary file names are recognized by DOS:

.SYS "System" file. These are executable by DOS, simply by typing the file name as a DOS command. .SYS files are not listed by the DIR command unless specifically requested. .SYS files are called "transients" since they are part of the set of DOS commands, but do not reside in memory at all times. .SYS files are listed in the directory in YELLOW.

.KIL "Killed" file. These are files which have been removed from active status by the KILL command. A .KIL file will be removed by COMPRESS, however it may be recovered before COMPRESS by using RENAME. .KIL files are not listed by the DIR command unless specifically requested. .KIL files are listed in the directory in RED.

All of the file types below are listed in GREEN in the directory.

.SRC "Source" file. These files contain ASCII text, such as the source code of an assembly program.

.BUF "Create Buffer" files. These files are created by the BUFF command and recalled with DRAW. They contain commands used to draw pictures.

.PIC "Picture" files. These files contain a dump of up to two megabytes of image memory. They are created with PICTURE and recalled with REFRESH.

.RLE "Run-Length Encoded" files. These files contain a compacted version of the data from image memory. They are created with IMPLODE and recalled with EXPLODE.

.ABS "Absolute" binary files. These files contain a dump of bytes from selected areas of memory. They are created with STORE and recalled with FETCH.

FILE NAME PATTERNS

DOS allows a "pattern" to be used in place of a file name under many conditions. A pattern permits a single command to affect several files at once, or permits a command to affect any file meeting a set of criteria. Depending on the command, using a pattern will either affect the FIRST file on a disk which matches the pattern, or ALL files which match. Details are given in the descriptions of the individual commands.

A pattern may consist of any combination of these items:

A primary name

A secondary name (example: .SYS)

A wild card "*" in either the primary or secondary name

A password

A drive number

The asterisk "*" performs a special function. If the primary name is an asterisk, it will match any file name. If the secondary name is an asterisk, it will match any secondary name. If either field CONTAINS an asterisk (in addition to other characters), the asterisk will match any single character in a file name. If the asterisk is in the last position of a field (in addition to other characters), it will match any set of zero or more characters.

If the primary name is blank, an asterisk is assumed to be inserted in place of the blank. This means that the following two patterns are equivalent:

*.SRC

.SRC

Either of these patterns would match any file whose secondary name is .SRC (a text file).

If the secondary name is blank, it will match any file EXCEPT a .SYS or a .KIL file. These files are never matched except when specifically asked for, by using a .SYS, .KIL, or .* pattern.

Some examples of patterns:

A*	Matches any file beginning with "A" except .SYS and .KIL files
A*.*	Same as "A*", but includes .SYS and .KIL
*.BUF	Matches any .BUF file
.	Matches ANY file

Patterns can be very convenient, but they should be used with caution. Suppose a program created a set of scratch files, and named them X1, X2, and so on. They could all be removed at once with the command

```
KILL X*
```

but this would also KILL any other files whose names began with the letter X.

TRANSIENTS

Transients, or transient programs, are the files which make up the set of commands DOS recognizes. By typing the name of a command, you tell DOS to search the disk for the file with the same name. If the file is found, it is loaded and executed, causing your command to be carried out.

This system of swapping commands in and out of memory as needed gives DOS great flexibility. The entire DOS need never reside in memory at once; only the current command is occupying space in memory. Further, it is simple for you (or Chromatics) to add commands to the set of commands DOS recognizes, by writing assembly language programs to carry out the command. Transients are stored on the disk with a secondary name of .SYS, and are not visible in the disk directory unless you specifically ask to see them.

This section discusses the transients, or commands, supplied with DOS. When typing in a command to DOS, the various parts of the command line must be separated by delimiters. Valid delimiters are:

SPACE

comma (,)

Certain control-characters and punctuation marks will also act as delimiters, but their use is not recommended since it would make the command line difficult to read.

If a command line contains several file names, delimiters must occur between the names. Only a SINGLE delimiter should be used to separate each pair of items on the command line; i.e., you should NOT type a comma followed by a space. This would usually cause a "Syntax Error" message.

DIR

Format:

DIR [<pattern>] [;<options>] RETURN

Where:

<pattern> is a file name, or a pattern which may contain wild cards.

<options> are described below.

The DIR (Directory) command lists the files in the directory of a disk. Several options can be specified to tell the DIR command which files you are interested in.

Typing DIR by itself will give you a list of most files on the disk whose password matches yours. Files with either a .SYS or a .KIL secondary name are not listed when you type DIR.

If <pattern> is included, only files matching the pattern are listed. Some examples of using a pattern with DIR are shown on the following pages.

If you want to examine the files of another user, you may enter that user's password as part of the pattern. It should be separated from the rest of the pattern by a dollar sign ($).

You may use the DIR transient, residing on one disk, to examine the files of another disk. This is normally done only if the second disk does not contain the DIR transient. It is accomplished by specifying the drive number of the disk whose directory is to be listed. This number is preceded by a slash (/).

In the examples below, and on the following pages, the RETURN key has been omitted, and a space has been used as a delimiter, so that the example will closely resemble what you will see on the 7900 screen.

DIR	List all except .SYS and .KIL files
DIR .*	List ALL files (including .SYS and .KIL)
DIR .SYS	List all .SYS files
DIR *.SYS	List all .SYS files (same as above)
DIR BR*	List all files whose names begin with the letters BR. Possible matches would be BR, BREAK, BROWN, etc.
DIR T*N	List all files whose names are three characters long, begin with T, and end with N (TEN, TON, etc.)
DIR $XY	List files under password "XY"
DIR .*$XY	List all files under password "XY"
DIR/2	List files on drive 2
DIR/1 /2	List files on drive 2, using the DIR transient living in drive 1
DIR/1 .KIL/2	List all killed files on drive 2, using the DIR transient from drive 1

You can also append some options to the DIR command, using a semicolon (;) to separate them from the pattern. These options are:

P List Public Files only, not the files under your password (if you logged in with no password, this is equivalent to listing the normal directory).

L Give Long version of the directory, including the disk name, address of next available space on the disk, address and length of each file, and "attribute" flags pertaining to each file.

S Give Short version of the directory, with file names only. Note: LONG version is default.

A List files stored under ALL passwords.

Examples:

DIR ;P	List public files
DIR .*;P	List all public files
DIR .*;L	List all files, with details
DIR/3 .*;LP	List all public files on drive 3, with details
DIR ;A	List files under all passwords (except .SYS and .KIL files)
DIR .*;A	List files under all passwords, including .SYS and .KIL files

NOTE: A SPACE must be present after the command DIR. No space is used between a pattern (if any) and the semicolon. This is illustrated in the examples above.

The disk directory is displayed in this form:

```
DISKNAME              Free Address: $nnnn      Free Length: $xxxxx
-------------------------------------------------------------------
Filename      Status   File     File    File Origin  Last Access
Prefix  Sfx  ........ Address  Length  Date   Time  Date   Time
-------------------------------------------------------------------
Sample12.SRC  w.......  $4000    $200
```

The disk is named "DISKNAME." The first free byte on the disk is byte number $nnnn, and the length of the free space located at that byte is $xxxxx bytes. (All numbers prefixed by the dollar sign are in hexadecimal, base 16.)

One file is listed in this directory. It is named Sample12, and has .SRC as a secondary name. The file begins at byte $4000 and occupies $200 bytes of the disk. It is write-protected (see below).

The "Status" column of a directory may show any of the following characters:

- **w** The file is write-protected.
- **d** The file is delete-protected and cannot be destroyed by COMPRESS.
- **e** The file is execute-only.
- **o** The file is odd length. A file will only occupy exactly the number of bytes it requires, unless it contains an odd number of bytes. In this case, a single extra byte of storage is used by DOS to cause the file to occupy an even number of bytes. Note the efficiency of this scheme, in comparison with other disk operating systems which use blocks of 128 or 256 bytes, regardless of the actual file length.
- **k** The file has been KILLed.

If your system contains the optional Real-Time Clock, DOS will also display time and date information in the directory. The last columns of the directory will show when a file was created, and when it was last accessed.

The "Free Length" entry in the directory always shows the length of the largest free space available on the disk. If this number approaches zero, the disk is getting full and should be COMPRESSed. See the COMPRESS command for details.

COPY

Format:

COPY <source> <dest> RETURN

Where:

<source> and <dest> are each file names, or file name patterns. Wild cards are allowed.

COPY produces a copy of a file, on the original disk or on another disk. The name of the copy may be the same as the original, or different. It is NOT legal to copy a file to the same disk and retain the same name, however.

If a pattern is used instead of a file name, all files matching the pattern are copied. This provides an easy way to transfer only .SYS files to a new disk, for example.

NOTE: If <dest> is located on the same drive as <source>, a wild card is not allowed in the secondary name.

NOTE: If the secondary names and the drive numbers of <source> and <dest> are identical, only ONE file is copied regardless of any wild cards in the file names. This rule, and the one above, are required to prevent DOS from copying a copy (of a copy of a copy...)

.KIL and .SYS files are not recognized by COPY unless specifically requested (see examples).

The new file produced by COPY will have the same password as the old file, unless the command line specifically changes the password (by providing a password on the destination name).

The new file produced by COPY will always have the same status as the old file (execute-only, delete-protected, etc.). See DIR for a discussion of status attributes.

Examples:

COPY AX BX	Make a copy of AX; call it BX, and put it on the same disk with AX.
COPY AX/1 /2	Copy AX from drive 1 to drive 2.
COPY AX/1 BX/2	Copy AX from drive 1 to drive 2, and call the copy BX.
COPY T*/1 /2	Copy all files beginning with the letter T, from drive 1 to drive 2 (except .SYS and .KIL files)
COPY *.SYS/1 /2	Copy all .SYS files from drive 1 to drive 2
COPY AZ BZ$aa	Make a copy of AZ; call it BZ, and give it password "aa".
COPY AZ$aa $bb	Copy file AZ from password "aa" to password "bb".
COPY .SRC .BUF	Copy all text files into .BUF files.

The destination disk should always be formatted before you try to COPY anything onto it. Formatting prepares a disk to receive data. (The FORMAT command is discussed after we complete the description of COPY.)

only the drive numbers are specified, and no file name ttern is given, a special full-disk COPY occurs. This pies all data from the source disk to the destination disk, cluding the disk name and the entire directory. This kind of PY is normally used to produce a backup copy of an entire disk. FULL-DISK COPY DESTROYS ALL DATA ON THE DESTINATION DISK.

ample:

COPY /1 /2	Copy the entire disk in drive 1 onto the disk in drive 2.

NOTE: A SPACE must occur between the COPY command and the source drive number. If drive 1 contains the COPY transient, the command above is equivalent to

COPY/1 /1 /2

In this case, the drive number of the disk containing the COPY transient is specified. The following command would NOT be legal:

COPY/1 /2

This command specifies the transient drive and the source drive, but does not specify the destination drive. The result is an "Argument Error" message.

When a diskette (floppy disk) is FORMATted, it is defined to be either single-density or double-density. A single-density diskette can hold up to 256,256 bytes and a double-density diskette can hold up to 509,184 bytes. The 7900 normally uses only double-density diskettes.

If you attempt a full-disk COPY between two disks which have different densities, a warning will be displayed:

Density mismatch. Continue (Y/N) ?

Press the "Y" key if you want to proceed. If you asked DOS to copy from a single-density diskette to a double-density diskette, the "density mismatch" will not be a problem, since all the data on a single-density diskette can easily fit onto a double-density diskette. If, however, you asked DOS to copy from a double-density diskette onto a single-density diskette, you will get an error after the disk has been halfway copied.

A similar situation arises if you attempt a full-disk COPY between a diskette and the hard disk. The capacity of a diskette is approximately 256K bytes or 512K bytes, depending on the density. The hard disk capacity is 10M bytes. You will see this warning:

Capacity mismatch. Continue (Y/N) ?

If you press the "Y" key, copying will proceed. BEWARE: If you asked DOS to copy from a diskette to the hard disk, EVERYTHING on the diskette will be copied and no errors will be displayed. However, the hard disk directory will now be a copy of the diskette directory, and will reflect a disk size of 512K bytes instead of 10 Mbytes. This will render 95% of the hard disk inaccessable until the hard disk is reformatted.

If you asked DOS to copy from the hard disk to a diskette, an error will occur as DOS attempts to write past the end of the diskette.

FORMAT

Format:

FORMAT [<name>] /<n> [;<opt>] RETURN

Where:

<name> is the name given to the disk being formatted. If omitted, the revision level of DOS is used to form the disk name.

<n> is the number of the disk drive containing the disk to be formatted. <n> is required.

<opt> are options, desribed below.

FORMAT initializes a disk, preparing it for data. A new disk must be formatted before it can be used. Formatting destroys all data on a disk. After a disk has been formatted, it contains a blank directory, and no files. The blank directory shows only the disk's name, its size, and allows DOS to determine the disk's density (single or double).

If <name> is specified, the disk is given this name. <name> may be one to eight alphanumeric characters. If <name> is not specified, the disk is named "DOSRevXX" where XX is the version of DOS which initialized the disk. A disk formatted by DOS 1.4 would be named "DOSRev14".

FORMAT prints the message:

Format drive n. Continue (Y/N) ?

where n is the drive number FORMAT will initialize. You are given this chance to abort the process. Press the "Y" key to continue, or any other key to abort. You may also insert a different diskette in the drive before pressing "Y".

When DOS begins formatting the disk, it prints

Formatting drive n.

If successful, control returns to DOS with no further messages. If an error occurs, such as a bad block detected on the diskette, error messages are printed and the FORMAT should be tried again. If errors continue, the diskette should be discarded.

Options may be included in the <opt> field, separated from the rest of the FORMAT command by a semicolon. Allowed options are:

;D double density

;S single density

;Y "Yes," proceed with formatting

The default format for diskettes is double density. If desired, a single density diskette may be formatted using the "S" option. A diskette created for single density may be used on any CGC 7900 system, and may possibly be useful in exchanging data with other computer systems using single density diskettes. However, the directory of any 7900 disk will be different from that used by other disk operating systems, so it will not generally be possible to insert a CGC 7900 disk into any other computer system. Nor is it generally possible for the 7900 to read a diskette created by another computer system.

Option "Y" prevents the system from asking the "Continue?" question above. If option "Y" is used, the disk or diskette will be immediately formatted with no questions asked. Option "Y" should be used with caution.

INITIALIZING A NEW DISKETTE

Formatting prepares a disk for data, but does not store anything on it. A freshly formatted disk has a blank directory. Here is a sample procedure which might be used to format a brand new diskette:

Load a disk containing the FORMAT command, and other transients, into drive 1. Load a blank disk into drive 2.

Enter the following:

```
FORMAT/1 NewDisk/2
```

and press RETURN. The FORMAT transient responds:

```
Format drive 2. Continue (Y/N) ?
```

Press the "Y" key. The formatting proceeds. When complete, and no error messages have been printed, you can assume the new disk is now formatted properly. It has the name "NewDisk". Now you may wish to copy the DOS transients onto this disk, so that it will be useable without referring to other disks. Enter:

```
COPY/1 *.SYS/1 /2
```

and press RETURN. All system transients (.SYS files) will be copied to the new disk.

RENAME

Format:

RENAME <file1> [<file2>] [;<opts>] RETURN

Where:

<file1> is the name of an existing file,

<file2> is the new name to be given to the file.

<opts> are options, described below.

RENAME alters a file's name. The primary name, secondary name, or both, may be altered in the RENAME process. <file2> must not specify a drive number, since the file itself does not move from the disk where it currently resides. If <file2> specifies a drive number, that number is ignored.

RENAME may also be used to change a file's attributes, such as write-protection (see below). If you want to change a file's attributes and leave its name the same, you may omit the <file2> name. It is NOT legal to omit both <file2> and <opts>. At least one of these must be present.

RENAME is not allowed to change a file's password. COPY should be used for this purpose.

If <file1> contains wild cards, any files matching the pattern are renamed. If <file2> contains wild cards, characters are pulled from the name of <file1> to substitute for *'s in <file2>.

If <file1> specifies only a primary name, all files with that name (regardless of their secondary name) are renamed. Likewise, if <file1> specifies only a secondary name, all files with that name are renamed (regardless of their primary name). This can result in more than one file having exactly the same name. If this occurs, you can rename the files again (giving them different names) by specifying the primary AND secondary name of the file to be renamed. If <file1> completely specifies a file name, RENAME will only act on one file.

RENAME will not affect .SYS or .KIL files unless you specify a secondary filename of .SYS, .KIL, or .* in <file1>. RENAME can change a .SYS file to a .KIL, and can revive a .KIL file into its original type. BE CAREFUL.

The option field may be used to alter the file's status attributes (see DIR). Allowed options are:

W	Write-protect
-W	Write-enable
D	Delete-protect
-D	Delete-enable
E	Execute-only

If no options are specified, the file retains its old attributes. Note: for security reasons, RENAME will not remove the execute-only status from a file. Once a file has the "e" status, that status may not be altered.

Examples:

RENAME AX BX	Rename file AX to BX, giving it the same attributes AX had.
RENAME AX BX;W	Same as above, but write-protect the file.
RENAME DATA OLDATA;WD	Rename DATA to OLDATA, write- and delete-protecting it.
RENAME ZOO;-W-D	Remove W and D attributes from the file ZOO.
RENAME SECRET;E	Make file SECRET execute-only.
RENAME XX/1 YY/2	File XX on drive 1 is renamed to YY; the "/2" is ignored.
RENAME XX	This produces "Argument Error."
RENAME AB* CD*	Any file beginning with "AB" will now begin with "CD."

RENAME can also change the name of a disk (the name given to the disk when it was FORMATted). Use this form of the command:

```
RENAME /2 Newname
```

The disk in drive 2 will be renamed to "Newname".

KILL

Format:

KILL <filename> RETURN

Where:

<filename> is the file to be KILLed.

The KILL transient changes a file's secondary name to .KIL (except if the file is a .SYS file, it cannot be killed by KILL).

KILL is used to remove a file from "active" status, and mark it for eventual destruction with COMPRESS (see below). After a file has been killed, it is still recoverable, but is not recognized unless specifically requested. The DIR command will not display killed files unless the .KIL or .* pattern is included. Most programs will ignore killed files.

.SYS files may be killed using the RENAME command, to change their secondary name to .KIL (this should only be done with great care).

After a file has been killed, it is still recoverable until the disk is compressed (see COMPRESS). A .KIL file may be recovered using the RENAME command, to change the secondary name to something other than .KIL (.BUF, .SRC, etc.). After a COMPRESS, the file is NOT recoverable.

If a pattern is used instead of a filename, ALL files matching the pattern are KILLed. Wild cards (*) may be included in the pattern.

Examples:

KILL DATA	Kill the file named DATA
KILL DATA/2	Kill DATA on drive 2
KILL DATA$XY/2	Kill DATA on drive 2, passworded "XY"
KILL X*	Kill any file beginning with "X"
KILL TEST	Kill any file with primary name TEST
KILL .SRC	Kill any file with secondary name .SRC

COMPRESS

Format:

COMPRESS [$<pw>] [/<d>] [;A] RETURN

Where:

<pw> is the two-character password of the files to be COMPRESSed. If omitted, the password under which you logged onto DOS will be assumed.

<d> is the number of the drive containing the disk to be compressed. If omitted, the disk which contains the transient is compressed.

The COMPRESS transient disposes of all files whose secondary name is .KIL (killed files). The space on the disk formerly occupied by these files is now available for use again. The disposed files are NOT recoverable. COMPRESS also reclaims the space formerly occupied by deleted files (see DELETE).

If a file is delete-protected, it will not be affected by COMPRESS even if it has been KILLed.

If the ;A option is given, ALL files are compressed regardless of their passwords.

It is advisable to make a backup copy of important files, or of the entire disk, before executing COMPRESS. If COMPRESS is interrupted by pressing Reset, by a power failure, or by removing the diskette during COMPRESS, all data on the disk may be lost.

COMPRESSing the hard disk (drive 3) may take several minutes.

Examples:

COMPRESS	Remove .KIL files
COMPRESS/2	Compress the disk in drive 2
COMPRESS/1 /2	Compress the disk in drive 2, using the COMPRESS transient from drive 1
COMPRESS $AB	Remove .KIL files under password "AB"
COMPRESS ;A	Remove .KIL files under ALL passwords

DELETE

Format:

DELETE <file> RETURN

Where:

<file> is the name of the file to be deleted.

DELETE removes a file from the disk directory. This process immediately frees up the disk space formerly occupied by the file. THE CONTENTS OF THE FILE ARE NOT RECOVERABLE.

The DELETE command is different from KILL. KILL simply renames a file with a .KIL secondary name, thus the file still exists. After a file has been deleted, however, the disk directory no longer shows the file's existance.

When the Disk Operating System is asked to create a new file, it always takes the largest currently available chunk of disk space. This means that it is not useful to DELETE a small file, since the small chunk of space formerly occupied by that file would never get used. DELETE is primarily useful for times when you need to remove a large file (one occupying over 25% of the disk, for example). DELETE allows you to reclaim the disk space without going through the (time-consuming) process of KILLing the file and then COMPRESSing the disk.

A pattern may be used in place of a file name. All files matching the pattern will be DELETEd.

Example:

```
DELETE OLDdata
```

The space formerly occupied by deleted files is reclaimed when the disk is COMPRESSed. If you DELETE several small files (going against the advice above), you will have to COMPRESS the disk to make that space available for use again.

If a file is delete-protected, it cannot be deleted.

BUFF

Format:

BUFF <file> RETURN

Where:

<file> is the name to be given to the file being created by BUFF.

The BUFF command stores the contents of the Create Buffer into a disk file. The secondary name .BUF is given to the file. If the file name already exists on this disk, the old file is automatically KILLed.

.BUF files can be called back into the Create Buffer with the DRAW transient.

Examples:

BUFF ANDWAX	Create a file called ANDWAX.
BUFF/1 FUDD/2	In this case the .BUF file is created on a drive other than the drive where the BUFF command resides.

When creating a picture to be stored as a .BUF file, the following hints may be helpful:

1. Turn on CREATE before transmitting any other commands which are necessary to set up the system for your picture. For example, if your picture will be drawn in the Bitmap, your .BUF file should include the "Overlay Off" and "Overlay Transparent" commands. Remember that the best .BUF files can be drawn directly from DOS, requiring no setup by the user.

2. Turn the cursor off at the start of your .BUF file. This makes a picture redraw faster.

DRAW

Format:

DRAW <file> RETURN

The file named <file> is called up from the disk and stored into the Create Buffer. <file> must have a .BUF secondary name.

After using the DRAW command, pressing the REDRAW key will cause the picture in the Create Buffer to be redrawn. Pressing XMIT will cause the Create Buffer contents to be sent out Logical Output Device 1, normally the RS-232 serial port.

Examples:

DRAW POKER	The file POKER.BUF is loaded into the Create Buffer.
DRAW/3 Flies/1	The DRAW transient from drive 3 is invoked to call up the file Flies.BUF from drive 1.

NOTE: For best results, you are advised to press the TERMINAL key (leaving DOS and entering the Terminal Emulator) before pressing REDRAW. If you do not press TERMINAL before REDRAW, you will find that you are still in DOS after your picture is redrawn, and anything you type may obliterate your picture. To re-enter DOS after the picture finishes redrawing, press SOFT BOOT and DOS. This will leave any picture in the Bitmap intact.

If <file> was created on a CGC 7900 containing a different number of image memory planes, the picture you generate may not look the same as the original picture.

APPEND

Format:

APPEND <file> RETURN

The APPEND command is similar to DRAW, except that the <file> specified by APPEND is added to the end of the Create Buffer, instead of replacing whatever was previously in the Create Buffer. By performing a DRAW followed by one or more APPENDs, several .BUF files may be concatenated. Then the BUFF transient may be used to store the entire series as a single file.

Example:

DRAW Part1	load part one of a picture
APPEND Part2	add the second part
APPEND Part3	and the third part
BUFF ALLofit	store the whole thing

See also BUFF and DRAW.

PICTURE

Format:

PICTURE <file> [;<n>] RETURN

Where:

<file> is the name of a file to be created by PICTURE. If a file with that name already exists, the old file is KILLed.

<n> is a hex number (see below).

PICTURE stores an image from Bitmap memory into a disk file. This requires up to two megabytes of storage, depending on the number of Image Memory planes installed in your system, and thus consumes a large amount of disk space. If the disk becomes full while PICTURE is storing data, an error occurs and no data is saved on the disk.

PICTURE also stores the contents of the Color Lookup Table so that the current colors on the Bitmap screen can be recreated later.

The file created by PICTURE has a .PIC secondary name.

Example:

PICTURE THIS

8 memory planes ⇒ 1M byte

4 " " ⇒ ½M byte + overhead (1K color LUT)

1 " plane ⇒ 1M bit = 128K bytes + overhead.

The optional argument <n> allows you to specify which planes of Bitmap memory will be stored. You can store only the planes which are applicable to your picture, and save disk space by not storing unneeded planes. <n> is a hexadecimal number between 0 and FFFF. Each bit in <n> which is SET corresponds to a plane which will be stored. The least significant bit of <n> corresponds to plane 0, the most significant bit to plane 15. If you enter more than four hex digits, only the last four are used.

Example:

```
PICTURE FRAME ;87H    =    1000,0111
```

The hex number 87 has bits 0, 1, 2, and 7 set. This command stores the four planes which are normally installed in a four-plane system. If your system contains only four planes, this example is equivalent to using PICTURE without the optional <n>.

> NOTE: Each plane stored by PICTURE occupies 128K bytes of a disk, or $20000 hex bytes. The "Free Length" entry in the directory must be enough to accomodate this length (plus 1024 bytes for the Color Lookup Table), or PICTURE will generate an error message.

Example:

```
PICTURE BOOK ;7
```

This example stores only planes 0, 1, and 2, a total of 385K bytes. If a picture was drawn on a four-plane system and the blink plane was not in use, this command would store all information necessary to reproduce the picture. Note that this file would fit on a floppy diskette, but if four planes had been stored, it would not fit.

For most applications, BUFF, DRAW and APPEND are much more efficient methods of storing images. See the preceding pages for descriptions of these transients. See also IMPLODE and EXPLODE.

REFRESH

Format:

REFRESH <file> RETURN

Where:

<file> is the name of a .PIC file to be brought in from the disk, and displayed in image memory.

REFRESH is the opposite of PICTURE. REFRESH brings in up to two megabytes of data from the disk and displays them on the screen. The Color Lookup Table is also loaded by REFRESH. See the description of PICTURE for details.

If the .PIC file specified in the REFRESH command was created on a CGC 7900 containing a different number of Image Memory planes, the image produce by REFRESH may not look exactly like the original image stored by PICTURE.

Example:

REFRESH Yourself

CHROMATICS D.F.

CGC 7900
Color Graphics Computer

Preliminary DOS Manual
(including Assembler
and Text Editor)

July, 1981

IMPLODE

Format:

IMPLODE <file> [;<n>] RETURN

Where:

<file> is the name of a file to be created by the IMPLODE command. If a file by that name already exists, the old file is KILLed.

<n> is a hexadecimal number (see below).

IMPLODE stores an image from Bitmap memory, similar to PICTURE. However, IMPLODE uses a data compression technique which can significantly reduce the amount of storage a picture requires. Like PICTURE, IMPLODE also stores the Color Lookup Table.

The file produced by IMPLODE has a .RLE secondary name.

The advantage of IMPLODE over PICTURE depends on the complexity of the image, and in extreme cases, IMPLODE can actually use more disk space than PICTURE. IMPLODE will display the number of bytes it stored, and will also display the number of bytes PICTURE would have used. You can then decide whether to try PICTURE instead.

<n> is a hexadecimal number which tells IMPLODE which planes to store. It acts exactly like the optional <n> argument in the PICTURE command. See the description of PICTURE.

Examples:

IMPLODE Baseball

IMPLODE CRT ;7

EXPLODE

Format:

EXPLODE <file> RETURN

Where:

<file> is the name of a .RLE file to be displayed in Bitmap memory.

EXPLODE is the opposite of IMPLODE. The data produced by IMPLODE will be expanded back into its original form and displayed in Bitmap memory. The Color Lookup Table will be restored to the colors it had at the time the picture was stored by IMPLODE. See the description of IMPLODE for details.

If <file> was created on a CGC 7900 containing a different number of Bitmap memory planes, the image produced by EXPLODE may not look the same as the original image.

Example:

EXPLODE Dynamite

STORE

Format:

STORE <file> <addr1> <addr2>

[+-<offset>] [@<exec>] [;<options>] RETURN

Where:

<file> is the name of the file to be created by STORE.

<addr1> and <addr2> are the starting and ending (hex) addresses of the range to memory to be stored.

<offset> is an address offset (hex) specifying the difference between the address the data was STOREd from and the address it will be loaded into. <offset> must be preceded by a + or - sign.

<exec> is the address at which execution of the data must begin (assuming the data is a program)

<options> are described below.

STORE creates a disk file containing all bytes from the range of memory <addr1> thru <addr2>, inclusive. The file created by STORE may contain an absolute binary image of memory, or it may be an image in executable form (readable by the DOS loader). STORE decides which type of file to create, based on the secondary name of <file>.

STORE will allow you to specify any secondary name. If you do not specify a secondary name, the default secondary name of .SYS is used and a load module is generated. If you specify .OBJ as a secondary name, a load module is also generated. Any other secondary name causes an absolute binary image to be stored into the file.

The arguments to STORE are affected by the secondary name of <file>:

If the file type is .SYS, the default value for <offset> is zero, and the default value for <exec> is <addr1>.

If the file type is .OBJ, the default value for <offset> and <exec> is zero. If the .OBJ file will be renamed to a .SYS file, <exec> must be specified at this time: a .OBJ file whose <exec> is zero cannot be executed.

If the file type is neither .SYS nor .OBJ, <offset> and <exec> are not allowed. An absolute binary file is generated which stores data in an unformatted form, storing just the bytes and no addressing information. This type of file may be given any secondary name, although .ABS is recommended.

A file created by STORE may be brought back into memory by FETCH or DEBUG. If STORE is used to create a .SYS file, the file may be executed directly by typing its name as a DOS command.

Examples:

```
STORE/1 HOUSE.ABS/2 4000 4FFF

STORE Program 14000 14FFF-10000@4400
```

Note that the + or - sign, and the @ sign, act as delimiters in the command line and should not be preceded by a space.

The option "P" makes a file "proprietary" by setting the execute-only status in the file's attributes:

```
STORE SECRETS 11C3C 12AF0-10000;P
```

The 7900 expects .SYS files to load and execute in the DOS transient program area (TPA) or the DOS buffer. The size of these areas is set with the 7900 "Thaw" command, but will normally be at least 16K bytes. The DOS areas begin at memory address $1C3C, and user programs should normally be arranged to run in this area.

The STORE transient also runs in the DOS area, and would overwrite any data in this area you are trying to STORE. That is the reason for the <offset> parameter. Data to be STOREd can be moved to a higher address with the Monitor "Move Memory" function, then the STORE command with an offset can be used to move the data back to its original addresses in the DOS area.

FETCH

Format:

FETCH <file> [<addr>] [+-<offset>] RETURN

Where:

<file> is the name of the file to be loaded into memory.

<addr> is the address where the data from <file> is to be loaded. <addr> is required if the file is NOT a .SYS or .OBJ file.

<offset> is a displacement, to be added to the normal load address of <file>. <offset> is required for a .SYS or .OBJ file.

FETCH is the opposite of STORE. It may be used to retrieve bytes saved by the STORE transient, or to load a .SYS or .OBJ file into memory.

FETCH will not load an execute-only file.

If the file read in by FETCH happens to over-write important areas of system memory, the system may hang.

Examples:

FETCH PGM.SYS/2+2000 Load the file PGM from drive 2, at memory addresses 2000 (hex) higher than it occupied when it was STOREd.

FETCH BYTES.ABS 1F000 Load the file BYTES into memory beginning at address 1F000 (hex)

DEBUG

Format:

DEBUG <filename> [<args>] RETURN

Where:

<filename> is the name of a .SYS file to be loaded into memory

<args> are the arguments expected by the .SYS file

DEBUG loads a .SYS file into memory, just as if the file had been executed by DOS. After loading the file, DEBUG jumps to the Monitor. If the file normally expects any arguments to be present on the command line, they may be entered as <args>.

DEBUG will not load an execute-only file.

To avoid having to use an offset when loading the .SYS file, DEBUG relocates itself to the top of the DOS Transient Program Area before loading <filename>. However, the TPA must be large enough to accomodate both DEBUG and your file. (DEBUG occupies about 512 bytes.) If necessary, the "Thaw" command can be used to change the DOS memory allocation. See the 7900 User's Manual for details.

Example:

DEBUG Process

DEBUG Gnats 6 12

In the second example, 6 and 12 are arguments to be passed to the program Gnats.

Since the current Monitor (version 1.1) reloads the stack pointer, it will not be possible for your program to execute a normal return to DOS. After using DEBUG to load your program, you may use the Monitor to trace program execution up until the point where your program attempts to return to DOS.

VERSION

Format:

VERSION <pattern> RETURN

VERSION displays the release date of the transients matching <pattern>. You may use VERSION to see whether you have the latest set of transients from Chromatics, or to indicate the release date of a transient when reporting a bug to Customer Service.

You must enter at least the disk number as <pattern>. Typing VERSION by itself causes an "Argument Error."

Example:

VERSION /2	Display the release date of all transients on the disk in drive 2.

SUMS

Format:

SUMS RETURN

SUMS performs a checksum of all PROMs in the 7900 system. This is normally used as a check on the integrity of a PROM, or to determine which version of firmware is installed in a system.

SUMS also displays the software revision level of the PROMs in your 7900 system, by searching for the ASCII string "VER#" in each PROM. If your system contains PROMs version 1.1 or higher (DOS version 1.4 or higher), SUMS will display the version number of these programs.

Example:

SUMS

XREF

›rmat:

XREF [^<options>] <file1>
[<file2>...<filen>] RETURN

here:

<file1>, <file2>, etc. are ASCII files containing an MC68000 assembly language program.

<options> are described below.

:REF is a program designed to be used with the Chromatics IC68000 Assembler (discussed in Section 3 of this manual). XREF ›roduces a cross-reference list of all labels in an MC68000 ıssembly language source file. The line at which a label is lefined is flagged with an asterisk (*).

<options> may include any of the following characters:

L	Transmit output to Logical Device 0 (normally the screen)
T	Transmit output to Logical Device 1 (normally the RS-232 serial port, assumed to connect to a printer)
P<n>	Print <n> lines per page (including 4 lines used as a header)
W<m>	Print lines up to <m> characters wide (<m> may range from 81 to 132)
-R	Don't cross-reference registers (A0, D1, etc.)

If <options> are omitted, the default is ^TP61W132+R. This causes listing to be directed to the printer, 61 lines per page, 132 columns per line, and registers are included in the XREF listing.

Example:

XREF Program

XREF ^LW85 Program	List on the screen, limit lines to 85 characters wide.

DOS ERROR MESSAGES

DOS will report errors which occur as a result of illegal commands, faults in the disk system, or programming errors. One of the messages below will be displayed when a DOS error occurs. When possible, DOS will display the drive number where the error occurred.

No index signal detected
No seek complete
Write fault
Drive not ready
Drive not selected
No track 000 detected

ID read error
Uncorrectable data error found during a read
ID address mark not found
Data address mark not found
Block not found
Seek error
No host acknowledgement
Diskette write protected
Data field error found and corrected
Bad track found
Format error

Invalid disk controller command
Illegal logical block address
Illegal function for the specified drive

Diagnostic RAM error

Disk controller not ready
Controller time out error
Unable to determine controller error
Undefined controller state
Controller protocol sequence error

Undefined load error state
Record cound error
Checksum error
Premature EOF during load
DOS buffer too small
Transient program size too small

End of file reached
File is write protected
Attempted to read thru density barrier
Attempted to transfer data on odd address

Unable to find requested file

Unable to create new file space

Unable to close requested file

Empty slot found
Unable to update the directory

No run address
Unable to find disk name
Argument error
Attempt to access a non-existant drive
Unable to initialize drive 1
Unable to initialize drive 2
Syntax error! Missing argument

Premature format termination
Error mapping routine not implemented
Unable to fetch this file
File is delete protected
File type error
File is execute only
File is to big to append
Insufficient stack size
/0 mode is not allowed in argument filenames
Undefined DOS error.

SECTION 2 - THE EDITOR

INTRODUCTION TO THE EDITOR

The Chromatics CGC 7900 Text Editor is a disk-based program used for creating and maintaining text files. It is primarily used in conjunction with the Assembler, for creating programs executable by the MC68000 processor. The editor is also good for working with other types of text files, such as correspondence or documentation. This manual was, in fact, written on a text editor.

The editor executes under DOS, the Disk Operating System, which was described in Section One of this manual. If the DOS prompt (a green asterisk) is not currently visible, press the DOS key. Enter your password and press RETURN (or simply press RETURN). Make sure that the system contains a disk which has the editor on it, the program EDIT.SYS. Then type:

EDIT RETURN

It may be necessary to specify the number of the disk drive containing the EDIT program, as:

EDIT/n

Where n is the number of the drive where EDIT.SYS resides. This will only be necessary if another drive, not containing the editor, has been in use.

When the editor expects input, it will prompt you in one of three ways. If the editor expects a command, the prompt is a 4-digit line number followed by a question mark. If it expects a line of text to be inserted into the file, the prompt is a 4-digit line number followed by an "I" and a question mark. If you are in MODIFY mode, the prompt is a number followed by the letter "M" and a question mark.

```
0000   ?      Command prompt

0000 I ?      Insert prompt

0000 M ?      Modify prompt
```

The line number is a pointer position within the file. The editor refers to lines by number, and maintains an internal pointer somewhere within the file. Commands are provided which will explicitly move the pointer around, and many commands will implicitly move the pointer. For example, if the pointer is on line 3, and you LIST lines 3 thru 20 of the file, the pointer is now on line 20.

> NOTE: When the edit pointer is at the beginning of the file, the line number is displayed as "B". When it is at the end, the line number is displayed as "E".

When you see the command prompt, you may enter any of the legal editor commands described in this manual. The INSERT command will take you out of command mode and put you into insert mode. When you see the insert prompt, anything you type will be inserted into the file at the current pointer position. Hitting the DELETE key will return you from insert mode to command mode.

Each of the commands in this section may be abbreviated to the smallest number of characters which will uniquely identify that command. For example, the OPEN command may be abbreviated to the letter "O", since no other command begins with that letter. However, the command PRINT can only be abbreviated to two characters, PR, so that it won't be confused with the PAGE command (both begin with "P"). In general it is safe to abbreviate commands to two or three characters.

INLINE

"INLINE" is the standard subroutine used by the editor for fetching a line of input from the keyboard. This routine is also used by other CGC 7900 programs, such as DOS. INLINE accepts a line of input, and allows editing, inserting and deleting characters, and overstriking characters. When the line is completed to your satisfaction, press RETURN. Note: the cursor can be anywhere on the input line when RETURN is struck, but the entire visible line will always be accepted as input.

The left and right arrow keys move the cursor around within the line currently being typed. The Home key moves the cursor to the left edge of the current line. The cursor position is used to determine where text will be inserted, or where other commands will take effect.

INLINE supports the editing commands printed in blue on the front of the cursor control keys: Insert Character, Delete Character, Clear Line, Clear to End Of Line, Recall Last Line. These blue functions are accessed by holding down the CTRL (control) key and pressing the indicated key.

Del Char (Delete Character) removes one character at the current cursor position. All characters to the right of the cursor move left one position.

Clear Line erases the line currently being typed.

Clear EOL (Clear to End Of Line) erases all characters from the current cursor position to the end of the line.

Recall (Recall Last Line) replaces the line currently being typed with the last complete line that was typed. This function is useful for repeating a command, perhaps altering it slightly with the other functions. Press Recall more than once to bring back earlier lines; this moves backward into the recall buffer. Press SHIFT with Recall to move forward in the recall buffer.

Ins Char (Insert Character) puts the routine into insert mode. The character under the cursor begins blinking, and any characters typed are now inserted, forcing characters to the right of the cursor to move out of the way. To leave insert mode, use one of the arrow keys to move the cursor. This places the routine in its normal (overstrike) mode, and any characters typed now will simply overwrite existing characters under the cursor.

INLINE is designed to be a general-purpose routine for ALL user input in the CGC 7900. The Appendix describes the calling sequence for INLINE, for users who wish to use it in their own programs. It is STRONGLY SUGGESTED that all programs use INLINE to accept input from the user. This means that all programs will support character editing as described above, and the user will become accustomed to using the same editing sequence for all program input.

EDITOR COMMANDS

This section discusses the commands accepted by the editor. Each of these commands may be entered at the command prompt.

In each command, one or more delimiters may be present to separate the various parts of the command. A delimiter may be a space or comma. For convenience, we will always use a space in our examples.

If the editor cannot interpret a command, or if for any reason an error occurs during a command, the command line is re-printed on the screen with the cursor positioned over the error. You may then edit the command line, using the INLINE editing functions on the cursor keypad. This avoids retyping the entire input line, and also illustrates exactly where the error occurred.

OPEN

Format:

OPEN <file> RETURN

Where:

<file> is the name of an existing disk file containing ASCII text. <file> is assumed to have the secondary name .SRC, unless a different secondary name is entered.

Before a file can be edited, it must be OPENed as an input file. The OPEN command searches for a specified file and returns an error if the file cannot be located. If the file does exist, OPEN simply returns to command mode. Note: OPEN does not actually cause any text to be read in from the file! See GET below.

The file is assumed to contain ASCII text. Each line of the file is terminated by a Carriage Return character, and no Line Feed. The editor will provide a Line Feed after each Return when LISTing or PRINTing the text.

Examples:

OP DOOR

OP WINDOW

OP ThatFile/2

GET

Format:

GET RETURN

GET <#> RETURN

Where:

<#> is a decimal number.

GET reads in text from the currently open input file (the file most recently specified by OPEN). If the GET command is used without an argument, enough text is read in to approximately half fill the available memory. If a <#> is specified, only that many lines are read in. The text is appended to the end of the text already in memory.

If the GET command causes the entire file to be read in, the message "End Of Input Data" is printed, and no more text may be read in from the input file.

If the GET command causes memory to be filled with text, the message "Workspace Full" is printed, and no more text is read in. It is now necessary to PUT some text back on disk, to make room for more of the file. See PUT and PAGE for examples. Note: Using GET without arguments is recommended, since it will never totally fill the workspace.

Examples:

```
GET

GET 30
```

LIST

Format:

LIST RETURN

LIST <#1> RETURN

LIST <#1> <#2> RETURN

Where:

<#1> and <#2> are line numbers.

LIST displays lines of text from the contents of memory. If no line numbers are entered, LIST begins at the current pointer position. If one line number is given, listing begins at that line and continues thru the end of the file. If two line numbers are given, listing begins at the first line number entered and continues thru the second line number.

LIST may be paused by typing a Control-S, and restarted with Control-Q. LIST may be stopped at any time by pressing DELETE.

The output from LIST is always directed to Logical Output Device 0, normally the screen or a part of the screen. Each line displayed by LIST is shown with its line number, for reference.

Examples:

```
LI

LI 200
```

PRINT

Format:

PRINT RETURN

PRINT <#1> RETURN

PRINT <#1> <#2> RETURN

PRINT performs the same function as LIST, but sends its output to Logical Output Devices 0 and 1. Since Logical Output Device 1 is normally connected to a printer, this produces a hardcopy of the lines listed. Unlike LIST, PRINT does not display line numbers in front of each line.

Examples:

PR

PR 100 190

INSERT

Format:

INSERT RETURN

INSERT <#> RETURN

Where:

<#> is a line number.

INSERT takes the editor from command mode to insert mode. While in insert mode, the prompt is in the form:

NNNN I ?

The "I" indicates that material is being inserted into text.

If <#> is entered with the INSERT command, insertion begins at the line specified. All lines from <#> up will move up in the file to make room for lines being inserted. If <#> is not specified, insertion begins at the current pointer position.

INSERT is the most important command in the editor. It allows you to enter text to create a new file. While in insert mode, any of the INLINE editing features may be used, such as insert and delete character.

Insert mode remains in effect until the DELETE key is struck, at which time the editor returns to command mode.

NOTE: The INSERT command causes the line numbers of part of the text in memory to be changed, as all lines past the insertion move up in memory. It is advisable to LIST the file after leaving insert mode, before performing any operations which are dependent on line numbers.

Examples:

```
IN

IN 15
```

If you enter a Mode code sequence (such as a "Set Color" command) into the input line, the sequence is displayed in compressed form, using special characters. It is not executed until you press the RETURN key. If you enter a tab character (CTRL I) into the line, it too is displayed but not executed. The Mode character resembles a double tilde (~), and the tab character resembles a right-pointing arrow. (These characters are taken from the "A7" character set, described in the CGC 7900 User's Manual.) Pressing RETURN will redisplay the input line with all Mode codes executing as they normally would when printed from a program. Tabs will be executed according to the current tab stop spacing in effect (normally 4 characters apart).

Using the up and down arrow keys or the Delete Line function, you can move from INSERT mode into MODIFY mode. Modify can also be entered by giving the MODIFY command, as discussed next.

MODIFY

Format:

MODIFY RETURN

MODIFY <#> RETURN

Where:

<#> is a line number.

MODIFY is the editor's most flexible mode. When you enter MODIFY, the editor's prompt is in the form

NNNN M ?

and is displayed in magenta. The current line is also displayed in magenta.

MODIFY allows you to use the INLINE editing features on existing text in memory. You can insert or delete characters using the labeled functions on the cursor keypad. When you have finished altering a line, you must press RETURN to store that line in its new form. If you move the cursor up or down using the arrow keys, the line you modified will NOT be stored, but will return to its previous condition.

Using the Insert Line and Delete line functions, you can move between MODIFY mode and INSERT mode at will. When in MODIFY, the prompt will be displayed in magenta and will be in the form "NNNN M ?" While in INSERT mode, the prompt is in yellow, and is in the form "NNNN I ?"

Pressing DELETE moves you back to command mode.

Examples:

MO

MO 25

When in MODIFY, as in INSERT, a special compressed form is used to display Modes, tabs, and other control-characters. In MODIFY, the line containing the cursor is always displayed in compressed form so that any control-characters in the line will be visible and may be edited. Other lines on the screen during MODIFY are displayed normally; only the line with the cursor is displayed in this special form.

DELETE

Format:

DELETE RETURN

DELETE <#1> RETURN

DELETE <#1> <#2> RETURN

DELETE removes a set of lines from the text. If DELETE is entered with no arguments, only the current line is deleted. (The current line is the line whose number is printed in the prompt.)

If DELETE is entered with one argument <#1>, the single line whose line number is <#1> is deleted. If DELETE is entered with both <#1> and <#2>, all lines within the range <#1> thru <#2>, inclusive, are deleted.

> NOTE: The DELETE command causes the line numbers of part of the text to be changed, as all lines past the lines deleted are moved down in memory. After a DELETE, it is advisable to LIST the file before doing any other operations which are dependent on line numbers.

FIND

Format:

FIND \<string>\ RETURN

FIND <#1> \<string>\ RETURN

FIND <#1> <#2> \<string>\ RETURN

FIND <#1> <#2> <N> \<string>\ RETURN

FIND locates a string. The range of lines to be searched, and the number of searches to perform, are specified in the command:

FIND with no arguments other than <string> begins searching at the current pointer position, and reports all occurances of <string> until the end of the file.

FIND with <#1> is the same as above, but begins searching at line <#1> rather than at the current pointer position.

FIND with <#1> and <#2> searches all lines from <#1> to <#2>, inclusive.

FIND with <#1>, <#2> and <N> begins searching at line <#1>, and terminates when it reaches line <#2> OR if it has found <N> occurances of <string>.

The backslash character "\" is used as a delimiter to define the search string. Any non-numeric character (except "[") could be used as a delimiter, provided it does not occur in <string>. The terminating delimiter (just before RETURN) is not required.

Examples:

FI \0\	Find all zeroes from the current pointer to the end of the file.
FI 1 999\.\	Find all decimal points in the file (through line 999)
FI 1 999 10\the\	Find up to 10 occurances of the word "the", between lines 1 and 999.

During a FIND, you may press CTRL S to pause the display, and then CTRL Q to continue. DELETE returns you to command mode.

SUBSTITUTE

Format:

```
SUBSTITUTE  \<string1>\<string2>\  RETURN

SUBSTITUTE  <#1>  \<string1>\<string2>\  RETURN

SUBSTITUTE  <#1>  <#2> \<string1>\<string2>\  RETURN

SUBSTITUTE  <#1>  <#2>  <P>
                        \<string1>\<string2>\  RETURN

SUBSTITUTE  <#1>  <#2>  <P>  <F>
                        \<string1>\<string2>\  RETURN
```

SUBSTITUTE performs a search-and-replace function. You can enter a number of options to specify exactly how the substitution takes place:

SUBSTITUTE with no arguments affects only the current line, and if <string1> is on that line, it is replaced by <string2>. Only one occurance of <string1> will be replaced.

SUBSTITUTE with <#1> affects only the line specified, and only one occurance will be replaced.

SUBSTITUTE with <#1> and <#2> begins at line number <#1> and continues thru line number <#2>. If <string1> is found on any line, it is replaced by <string2>, but only the first occurance of <string1> per line will be affected.

SUBSTITUTE with <#1>, <#2> and <P> affects all lines from <#1> to <#2>, inclusive, and replaces <P> occurances per line.

SUBSTITUTE with <#1>, <#2>, <P> and <F> affects all lines from <#1> to <#2>, inclusive, replaces <P> occurances per line, but begins at occurance <F> on each line.

The backslash "\" is used as a delimiter to define the beginning and end of each string. Any non-numeric character (except "[") could also be used as a delimiter, provided it does not occur in either <string1> or <string2>.

The terminating delimiter (just before the RETURN) is not required.

Examples:

SU \A\B\	Change the first occurance of "A" to "B" on the current line (if any).
SU 54\123\321\	On line 54, change "123" to "321" (one occurance at most)
SU 100 200\me\I\	Change "me" to "I" everywhere between lines 100 and 200 (not more than once per line).
SU 100 200 99\me\I\	Same as above, but up to 99 times per line (effectively changes all occurances on each line in the range).
SU 1 100 1 2\this\the\	Between lines 1 and 100, change the second occurance "this" to "the" on each line.
SU 1 100 99 2\this\the\	Same as above, but changes all occurances EXCEPT the first occurance on each line.
SU 10 55.\./.	Change backslash to slash using a period as a delimiter), once per line, between lines 10 and 55.

SUBSTITUTE displays each line it changes. To abort the SUBSTITUTE process, press the DELETE key. You will return to command mode, and any lines which have not already been displayed by SUBSTITUTE will not be affected.

SUBSTITUTE can be very destructive, if not used carefully. It is good practice to use FIND before SUBSTITUTE, to see exactly what will be affected by SUBSTITUTE. For example,

FI 1 99 1\A\	Find the first occurance of "A" and display the line.
SU \A\B\	Change "A" to "B" at this line

By using the Recall Last Line function, you can repeat these two commands as often as required, examining each occurance of "A" before changing it to "B".

LAST

Format:

LAST RETURN

Each time the command prompt is printed, the current pointer position is memorized. The LAST command moves the pointer to its most recent location in memory. Using LAST more than once will flip back and forth between the current location and the most recent location.

Here is a sample session with the LAST command:

7900's Output	User's Input
0010 ?	LIST 20 30
(Lines 20 thru 30 are listed)	
0030 ?	LAST
0010 ?	LAST
0030 ?	

PUT

Format:

PUT RETURN

PUT <#1> RETURN

PUT <#1> <#2> RETURN

PUT removes text from memory, and writes it back to the disk.

If PUT is entered with no arguments, all text in memory is written out to the disk. If one argument <#1> is entered, all lines from the beginning of text thru line <#1> are written out to the disk.

If two arguments are entered, only lines between line number <#1> and line number <#2> are written out to the disk.

PUT is primarily useful for dividing up a file into smaller files. By doing a PUT followed by a CLOSE, a new file is created which contains only the lines which were PUT.

PAGE and EXIT are more general-purpose commands for ending an editing session.

Examples:

```
PU

PU 60 90
```

CLOSE

Format:

CLOSE RETURN

CLOSE <filename> RETURN

CLOSE enters the output file into the disk directory, and closes the file. If CLOSE is entered without a <filename>, the new file has the same name as the old file; the editor will then automatically KILL the old file.

If a <filename> is specified, the new file has that name. The file will have .SRC as a secondary name unless you specify a different secondary name.

Once a file has been closed, either by CLOSE or by EXIT, it exists on the disk and can be re-opened by OPEN as an input file.

Examples:

CL

CL MIND

CL NEWFILE

PAGE and EXIT are more general-purpose commands for ending an editing session.

PAGE

ormat:

PAGE RETURN

The PAGE command is used when editing large files. If a file is so large that it cannot all fit into memory at once, it is necessary to bring in a portion of the file, edit that portion, then go on to the next. PAGE dumps all of the text in memory back to the disk, then brings in enough text from the input file to half fill the available memory.

PAGE is thus equivalent to doing a PUT of all text in memory, followed by a GET.

Example:

PA

DRIVE

Format:

DRIVE <N> RETURN

The output from the editor normally goes onto the same disk from which the input file was read. You may alter this with the DRIVE command, forcing the editor's output onto another disk. DRIVE causes the editor to create a new output file, on the specified disk.

Example:

DR 2

If you have already written some text to the output file, using PAGE or PUT, DRIVE is not allowed until you CLOSE the currently open output file. (If it were allowed, the text you had written to the disk would be lost.)

EXIT

Format:

EXIT RETURN

EXIT <filename> RETURN

EXIT is the proper way to end an editing session. EXIT first PUTs all text in memory onto the disk. Then it performs a series of GETs and PUTs (if necessary) to insure that the entire input file has been written to the output file. When no text remains in the input file, the output file is closed.

If EXIT is entered with no <filename>, the output file has the same name as the input file. The editor then automatically KILLs the old input file.

If EXIT is entered with a <filename>, the output file is given that name. The file will have .SRC as a secondary name unless you specify otherwise.

Example:

```
EX

EX RAMP
```

ABORT

Format:

ABORT RETURN

The ABORT command ends an editing session, but does not close the output file. If any changes had been made in the file, the changes are lost.

ABORT is used if you decide that you don't want to alter the file after all. If you had been using the editor simply to examine a file (rather than making changes), ABORT would be the logical way to end the session.

Example:

AB

NOTE: Pressing the DOS, MONITOR, or TERMINAL keys will also result in leaving the editor. To re-enter the editor after an abort, execute the key sequence

SHIFT USER W

this will "warm-start" the editor, with the data in memory intact.

SECTION 3 - THE ASSEMBLER

INTRODUCTION TO THE ASSEMBLER

The Chromatics MC68000 resident assembler is used to produce machine-readable object code from assembly language source files. The assembler executes under the Chromatics Disk Operating System, described in Part 1 of this manual.

The full MC68000 instruction set is supported by the assembler. In this manual, it is assumed that you understand the MC68000 processor architecture, as described in the Motorola MC68000 User's Guide (available from Chromatics). The examples provided in this manual are intended only to demonstrate proper syntax, and do not necessarily show useful programming techniques.

It is assumed that you are familiar with basic operating techniques used in the CGC 7900; if not, consult the CGC 7900 User's Manual before attempting to use the assembler.

ASSEMBLER COMMAND LINE

Format:

ASMB [^<options>] <file1> RETURN

ASMB [^<options>] <file1>
[^<options>] <file2> ... RETURN

Where:

<file1>,<file2>, etc. are the names of source files to be assembled

<options> are characters which specify assembly options (see below)

The ASMB command invokes the file ASMB.SYS, the Chromatics MC68000 resident assembler. The ASMB command must be entered at the DOS prompt, the green asterisk (*).

The assembler expects its input files to be in ASCII text form. Input files must have the secondary name .SRC.

The assembler produces an output file of type .SYS, which may be directly executed by DOS. If you enter the command line

ASMB PROGRAM

and press the RETURN key, the file PROGRAM.SRC would be assembled and an output file named PROGRAM.SYS would be produced. This .SYS file can be directly executed by typing its name as a DOS command:

PROGRAM

When the assembler closes its output .SYS file, any old .SYS file with the same name is KILLed. Thus, a program can be edited, assembled, re-edited and re-assembled any number of times, but only the most recent version of the source and object code will be active on the disk.

Several options are available to control the assembly and the output listing. The option field is indicated by the "carat" character, ^. The options may be entered in any order.

C	Close output file, producing an executable program file as output.
-C	Do not close output file.
T	Type listing on printer (through Logical Output 1).
-T	Display listing on screen (through Logical Output 0).
L	Follow LIST/NOLST commands in source file. The listing is on until a NOLST is encountered.
+L	Force listing ON regardless of source file.
-L	Force listing OFF regardless of source file.

The default conditions for options are C and L. If no options are specified, this will result in the output file being closed, listing on the 7900 screen, and listing being controlled by LIST or NOLST commands in the program source file.

If you do not specify any options, you must not enter the carat (^).

All options are valid at the start of the command line, before the first file name <file1>. You may repeat any of the "L" options prior to other files, in order to control listing of the various files individually.

Examples:

ASMB TEST	Assemble the file TEST, close output file, listing on the screen. Output file is named TEST.SYS.
ASMB ^-C TEST	Assemble, list to screen, do not close output file.
ASMB ^-L TEST	Assemble and close, no listing.
ASMB ^T-C+L TEST	Assemble, list on printer, do not close output file, listing ON.

ASMB ^T-L TEST1 ^+L TEST2

The last example assembles the program which is contained in both files, TEST1 and TEST2. The output file is closed, and is given the name TEST2.SYS. This file is created on the same drive which contained the file TEST2. Listing is supressed for file TEST1 and is forced on for TEST2. (Options other than the "L" family would not be valid prior to TEST2.)

If a program is contained in more than one file, each file must have its own END statement.

SOURCE FILE FORMAT

he text editor is used to create program source files, which re processed by the assembler. Source files have a secondary ame of .SRC, and are stored in ASCII text form.

'he general form of a line in the source file is:

[<label>] <instr> [<operands>] [<comments>]

The four fields may be separated by spaces or tabs.

<label> is optional. If included, it must be alphanumeric, from one to twenty-four characters. Other lines of code may refer to this line by referencing <label>.

<instr> is required. It must be a legal instruction mnemonic, as defined in the MC68000 instruction set; or it may be a pseudo-instruction recognized by the assembler. This manual includes descriptions of the pseudo-ops available. Consult Motorola literature for details on the MC68000 instruction set.

<operands> are the arguments to the instruction. Different instructions require different arguments.

<comments> may be included on any line, at the programmer's convenience. It may be useful to use a color-code at the beginning of a comment, to distinguish different sections of a program.

In addition, any line beginning with an asterisk (*) is ignored, and whole-line comments may be included in this way.

The first (or only) source file in a program will generally begin with an ORG statement, to initialize the Internal Program Counter (IPC). Each source file must end with an END statement.

LABELS

Labels always begin in the first column (character position) of a line of code. If the first character in a line is a space or tab, the line does not contain a label.

A label may contain any upper or lower case letter, A-Z or a-z, or digits 0-9. It may contain up to 24 characters, and all characters are significant. Upper and lower case characters are distinct; that is, the labels AB and Ab are considered different. A label must begin with a letter.

A label is assigned the value of the IPC at the instruction where the label occurs, unless the instruction is EQU or SET (see Pseudo-Instructions).

Certain reserved words have special meanings to the assembler, and may not be used as labels. They are:

D0	D1	D2	D3	D4	D5	D6	D7
A0	A1	A2	A3	A4	A5	A6	A7
SP	USP	CCR	SR	IPC			

It is possible, but not wise, to use an instruction mnemonic as a label.

INSTRUCTIONS

The instruction field is separated from the label by a space or a tab. If a line does not contain a label, the instruction must be preceded by a space or tab to separate it from the beginning of the line.

An instruction will either be a member of the MC68000 instruction set, or a pseudo-instruction which is recognized by the assembler as an order to do something.

If an instruction is a legal MC68000 op code, it may have a data size associated with it. Certain MC68000 instructions can operate on different data sizes: Byte (8 bit), Word (16 bit), or Long (32 bit). For these instructions, the desired data size may be specified by appending the characters ".B", ".W" or ".L" to the op code. The default data size of Word (16 bit) is assumed if the data size code is omitted.

If an instruction does not have a variable data size associated with it, the characters B, W or L may NOT be appended.

Examples:

```
MOVE    D1,D2       Word size is assumed by default.

MOVE.B  D3,D4       Byte size is declared.

LEA     (A1),A2     Long word size is required for this
                    instruction; a data size code would
                    not be permitted here.
```

OPERANDS

The operands, if any, follow the instruction. Operands are separated from the instruction by a space or tab. If more than one operand is present, they are separated by commas.

Most instructions have one or two operands. Instructions which use two operands consider the first to be a "source" and the second to be a "destination", as:

```
MOVE.W  D1,D5       Move 16-bit data from D1 to D5.

ANDI.L  #$7F,(A2)   Get 32-bit data from the address
                    pointed to by A2; AND it with
                    immediate data $7F (hex); then put
                    the result back where we got it.

SUB.W   D0,(A3)+    Subtract D0 from the 16-bit word
                    pointed to by A3; put back the
                    result, then increment A3 by 2.
```

The same terminology of "source" and "destination" is applied to compare operations. Note that the "destination" is compared to the "source", not the other way around:

```
CMP.L   A0,A1       Is A1 greater than A0?
BGT     Somewhre    If YES, do the branch.
```

COMMENTS

Any line may contain a comment field. The comment field may be separated from the remainder of the line by a "Set Color" command, which will cause the comment to be displayed in a different color from the program code. The color code is not necessary in most cases (except as mentioned in the note below), but it will improve the legibility of your programs. Color codes in a source line are never transmitted to a printer.

To insert a color-coded comment into a line of program source using the 7900 editor, first separate it from the rest of the line by a tab (CTRL I) or spaces. Then press SET, followed by a color key. When displayed by the editor in response to LIST, the color will be executed as typed. When in MODIFY or INSERT mode, the line containing the cursor is always displayed in compressed form, using special symbols for control-characters. A typical statement might normally appear as:

```
LABEL   EQU     value   This is a comment
```

If a color code had been typed before "This", the comment field would appear in that color. In MODIFY mode, or when entering the line in INSERT mode, the same line would appear as:

```
LABEL-EQU-value-~C6,This is a comment
```

The tab character is abbreviated with a right-arrow symbol, and the Mode character is abbreviated with a double tilde. This example uses color number 6, or yellow, as the comment color. You may wish to define a Function Key to be the equivalent of a "Set Color" command.

If a color code (or any Mode sequence) is present in a source line, the assembler assumes that it delimits a comment, and does not attempt to parse any items beyond the Mode character.

> NOTE: Certain op-codes will accept an indefinite number of arguments (Example: Define Constant, DC). In these cases, the assembler will continue to parse arguments from the source line until it reaches a Mode character or a carriage return. If you include a comment in a source line of this type, you MUST delimit the comment with a Mode code, or the comment will be assembled! Note that the END pseudo-op also accepts a variable number of arguments (zero or one expressions). A comment on an END statement should be delimited with a color code.

INSTRUCTION EXAMPLES

This section discusses the instructions available in the MC68000 instruction set, and provides examples.

ARITHMETIC

Format:

<op-code.size> <source>,<dest>

Add, subtract, multiply, divide, and negate are provided. Multiplication and division may use signed or unsigned arithmetic. BCD operations of add, subtract and negate are provided. Operations may use the X (extend) bit of the condition codes for multiprecision operations. CLR sets an operand to zero. Test, and Test-And-Set, set the condition codes according to an operand's value. TAS is also used to synchronize processors in a multiprocessor system.

Examples:

ADD.L	(A1),D2	Add the 32-bit data pointed to by A1, to the contents of D2; results in D2.
MULU	D3,D4	Multiply, with unsigned numbers, D4 by D3; results in D4.
SUB.W	#5,D1	Subtract 5 from the lower 16 bits of D1; put the result in D1.
DIVS	D5,D0	Divide D0 by D5 using signed numbers. Put the result in D0. (See Motorola literature for details of this instruction.)
CLR.B	(A3)	Zero the byte whose address is in A3.
TST.W	D4	Set condition codes according to the low 16 bits of D4.

COMPARE

Format:

```
CMP.<size>    <source>,<dest>

CHK           <bound>,Dn
```

<source> is subtracted from <dest> and condition codes are set accordingly. The conditional branch instructions are arranged so that a "Greater Than" condition tests true if <dest> is Greater Than <source>.

Neither <source> nor <dest> is altered by a compare.

CHK causes a processor trap to occur if the contents of register Dn are less than zero or greater than <bound>.

Examples:

```
CMP.L    (A3),D1     Compare D1 to 32-bit data at
                        address A3,
BLT      Label1      Branch here if D1 < (A3)
BGT      Label2      Branch here if D1 > (A3)
                     Else D1 = (A3) !  (Profound...)

CHK      #$100,D0    trap if D0 > 256.
```

LOGICAL

Format:

```
<op-code.size>    <source>,<dest>
```

Logical operations of AND, OR, Exclusive OR, and NOT are provided.

Examples:

```
ORI.W   #1,(A6)    OR immediate data (1) with the
                   word whose address is in A6.

AND.L   D4,D3      Logical AND D4 with D3 (32 bits),
                   results in D3.
```

SHIFT AND ROTATE

Format:

```
<op-code.size>    <count>,<dest>
```

Arithmetic and logical shifts and rotates are provided. The shift count may be specified by immediate data or by a register. If the effective address of the destination is in memory, only one bit position at a time may be shifted. Rotates may use the X (extend) bit.

Examples:

```
ASR     (A0)+      Shift the data pointed to by A0,
                   one bit right. Word size is implicit
                   since no other size is allowed here.

ROL.L   #3,D1      Rotate D1 3 bits left, using all
                   32 bits of data in the register.
```

BIT OPERATIONS

Format:

<op-code> <bit no.>,<dest>

A single bit may be tested and/or modified in a register or memory location. The condition codes are always set according to the state of the bit BEFORE any modification. If <dest> is a register, any of the 32 bits may be tested. If <dest> is an effective address in memory, the operation may only be byte size, and therefore only bits 0 through 7 may be tested.

Examples:

BTST	#0,StatWord	Check bit 0 in memory location StatWord. Set condition codes.
BCHG	#18,D5	Set condition codes according to bit 18 in D5, then flip that bit.

EFFECTIVE ADDRESS

The effective address of an operand may be computed and loaded into an address register, or pushed onto the stack.

Examples:

PEA	RetnHere	Push the effective address of the label RetnHere onto the stack. The next RTS (Return from Subroutine) will cause a jump to this address.
LEA	Displ(A0,D1.L),A6	Load the effective address found by adding A0 to D1 and to the value Displ; put the result in A6.

MOVE DATA

Format:

MOVE.<size>	<source>,<dest>
MOVEQ.L	#<immed>,Dn
EXG	Rn,Rn

Move operations transfer data between registers and/or memory. MOVEQ (Move Quick) is a special form which will load small data into the low byte of a data register, sign-extended into the upper three bytes. Using MOVEQ, <immed> must be between -128 and +127. EXG exchanges the 32-bit contents of two registers.

The MOVE instruction is very flexible, as all addressing modes may be used for <source> and most modes are also valid for <dest>.

Examples:

MOVE.L	D1,(A3)+	Move 32-bit data from D1 to the address pointed to by A3. Postincrement A3.
MOVE.B	#' ',D7	Put a space character into the low byte of D7.
MOVEQ.L	#$F,D0	Move a hex F (15) into D0. MOVE would work here, but would take extra object code.
EXG	D1,D5	Exchange register contents.

MOVE MULTIPLE

ormat:

MOVEM.<size> <Register list>,<dest>

MOVEM.<size> <source>,<Register list>

Move Multiple stores several registers into memory or retrieves them with a single instruction. <size> of word (.W) or long (.L) is allowed. <Register list> may include the names of data or address registers. A hyphen implies that all registers within the hyphenated list should be saved, while a slash implies that only the listed registers should be saved:

D1-D7 registers D1 thru D7, inclusive

D5/A1/A3 registers D5, A1, and A3 only

The registers must be specified in ascending order, with data registers listed first. MOVEM makes it easy to "push" and then "pop" a set of registers for temporary storage.

Examples:

MOVEM.L D0-A6,-(SP) Push all registers-D0 thru D7 and A0 thru A6 onto the stack (pointed to by SP).

MOVEM.L (SP)+,D0-A6 Restore these registers.

MOVEM.W D5-D7/A0,Storage Store registers A0, D5, D6 and D7 into words of memory beginning at label Storage.

BRANCH, JUMP

Format:

<op-code> <location>

JMP is an unconditional transfer to an absolute address. JSR is a subroutine call to an absolute address. There is no instruction to conditionally transfer to an absolute address.

BRA and its variations are relative transfers. The assembler computes the relative distance between the current IPC and the destination operand, and enters this as the relative displacement in object code. A branch must always reference a label within plus or minus 32767 bytes of the current IPC. (See also "short branch" below.)

Branches may be conditional or unconditional. Conditional branches are in the form Bcc, where cc is the condition under which the branch is taken. For example, BEQ will branch if the condition codes show the last comparison came out EQual (zero).

Valid conditional tests are:

CC	carry clear	LS	low or same
CS	carry set	LT	less than
EQ	equal	MI	minus
GE	greater or equal	NE	not equal
GT	greater than	PL	plus
HI	high	VC	no overflow
LE	less or equal	VS	overflow

The conditional tests PL, MI, GE, LT, GT, and LE treat operands as sign-extended numbers. If the high bit of the operand is set, the operand is negative. The other conditional tests treat operands as unsigned numbers. Motorola MC68000 processor literature lists the flags examined by each of the conditional tests above.

A branch may be in the form BRA.S, where the "S" implies "short". Since these are relative branches, a short branch may only be used to transfer to a location within plus or minus 127 bytes of the current PC. Using short branches whenever possible saves time and space. The assembler will automatically convert a branch to a short branch if possible. If the program contains a short branch to an address more than 127 bytes away, an error occurs.

Branches to a subroutine (BSR and its variations) push the return address onto the stack. A short form of subroutine call (BSR.S) is also provided. A RTS (Return from Subroutine) pops this address off the stack.

Examples:

JMP	Overhill	Unconditional transfer to label Overhill.
BRA	Faraway	Unconditional Branch to label Faraway.
BRA.S	Nearby	Unconditional Short branch to label Nearby.
BLT.S	Wasless	Short, conditional branch to label Wasless, occurs if condition codes show the last comparison came out "Less Than" (LT).
BSR	Subpgm	Call to a subroutine.
JSR	Farsub	Call to a subroutine which is more than 32767 bytes away, or which is accessed through an absolute address (such as a jump table).

DECREMENT AND BRANCH

Format:

```
DBcc   Dn,<location>
```

The DBcc instruction will implement a loop in a very simple fashion. It first tests the condition codes specified by cc. If this test is true, the instruction falls through (does not branch or perform any other operation).

If the test is NOT true, the lower word of data register (Dn) is decremented by one. Then if the lower word of the register does not contain a -1, the branch to <location> is taken. <location> must be within a 16-bit displacement of the DBcc instruction.

The loop will therefore continue until either (1) the test in cc is true, OR (2) the lower word of Dn contains -1.

Example:

```
        MOVE.W  #1000,D1    Loop up to 1001 times
Loop    TST.B   (A2)+       Is memory = 0 here?
        DBEQ    D1,Loop     If not, look again until
                            D1=-1 or the test came out
                            "EQ", meaning (A2)=0
```

A special form of this instruction is DBRA. No condition codes are tested by DBRA; the loop simply continues until the lower word of Dn is -1.

SET

Format:

```
Scc  <dest>
```

The byte specified by <dest> is set true or false, according to the result of the conditional test cc. If cc is true, <dest> is set to all ones. If false, <dest> is cleared.

Example:

```
SEQ     D4     If the zero flag is set, set D4 true.
```

See also TST (Test) and TAS (Test-And-Set), discussed with the arithmetic operators.

SYSTEM CONTROL

These instructions are used to control overall system operation or to set system flags.

RESET	Reset external hardware.
RTE	Return from exception (interrupt).
STOP	Stop execution and load status register.
TRAP	Load trap vector and execute routine.
TRAPV	Trap on overflow.

The MOVE, ANDI, ORI, and EORI instructions may be used to modify the SR (status register) or the CCR (condition code register).

> NOTE: All current 7900 software runs in Supervisor state. If your programs modify the SR, they should maintain the Supervisor bit SET at all times, or not alter this bit.

INSTRUCTION TYPES

The MC68000 instruction set includes certain op-codes which can be varied according to the addressing mode in use. These instructions are:

ADD	(add)
AND	(logical AND)
CMP	(compare)
EOR	(logical Exclusive OR)
MOVE	(move data)
NEG	(two's complement)
OR	(logical OR)
SUB	(subtract)

The allowed variations are selected by appending one of the following characters to the basic instruction:

A	Address. Destination register must be A0-A7, and byte size (.B) is not allowed.
I	Immediate. The size of the immediate data is (if possible) adjusted to the size specified by the instruction.
M	Memory. The CMPM form requires that source and destination operands must use postincrement (An)+ addressing mode.
Q	Quick. Allows small data to be operated on using a special, short form of the instruction. Size limits depend on the instruction.
X	Extend. Used for multiprecision operations.

In the case of the variations A, I and Q, the assembler automatically chooses one of these three if the program does not specify one. The program may always specify a variation, in which case, the rules for that variation must be followed. It is useful to explicitly specify the variation when writing well-documented code.

EXPRESSIONS

Many types of assembler instructions may contain arithmetic expressions. An expression is evaluated according to the standard rules of algebra. Spaces must not be typed within an expression, since the space character is used to delimit fields in the source program line.

The following operators are provided, and are listed in the order in which they are evaluated:

Unary minus (a minus sign with nothing in front of it)

Logical AND (&), logical OR (!)

Multiply (*), divide (/)

Add (+), subtract (-)

Operators with the same precedence are evaluated from left to right. Parentheses may be used to override the normal precedence, if desired.

Numbers are assumed to be decimal, unless preceded by a dollar sign ($) in which case they are interpreted as hex. ASCII constants (up to 4 characters long) and symbols may also be used in expressions. All intermediate results are stored in 32-bit form, and an 'O' error results if an expression overflows.

Examples:

```
9876+$A9B

BABY-grand*(PIANO/88)

'This'!'That'&THEM
```

PSEUDO-INSTRUCTIONS

Pseudo-instructions, or pseudo-ops, are instructions to the assembler. They affect the way the assembler generates object code. They do not, in general, generate code themselves. (The DC pseudo-op is an exception, and does generate code.)

ORG (ORIGIN)

Format:

```
ORG    <expression>

ORG.L  <expression>
```

ORG is used to set the "origin" of the code generated by the assembler. ORG loads a value into the Internal Program Counter (IPC). When using the (default) ORG form, all absolute addresses are assembled in "absolute short" form, which confines them to values between $0000 and $7FFF, and between $FF8000 and $FFFFFF.

In the CGC 7900, the "long" ORG.L is more commonly used. ORG.L forces all absolute addresses to long form, which allows access to the entire 7900 address space ($000000 to $FFFFFF).

(The exception to this rule occurs if a symbol name is suffixed with a ".W" or ".L" specifier. In this case, the specifier is used and the ORG type is overridden.)

Examples:

```
ORG.L   PROGSTART

ORG.L   $1C3C
```

DOS expects programs to run in the DOS Transient Program Area, which begins at $1C3C. Your programs should normally be ORG'ed at this address.

ORG may be used in several places during a program, if desired.

EQU (EQUATE)

Format:

```
<label>   EQU   <expression>
```

EQU equates a label to a value, the value of <expression>. <label> is required in this context, for the EQU statement is meaningless without a label. <expression> may contain constants or other labels which have been previously defined in the program.

Examples:

```
HexNum   EQU   $0A0A

Gold     EQU   Silver+32

HERE     EQU   IPC
```

The last example equates the label "HERE" to the value of the assembler's Internal Program Counter. The IPC always has the value of the address for which code is currenty being generated. Note the equivalence of the following two pieces of code:

```
SUBRT  MOVE.L   D1,D3
```

and,

```
SUBRT  EQU      IPC
       MOVE.L   D1,D3
```

Either of these two sections would generate identical object code if used in the same place in a program.

SET

Format:

<label> SET <expression>

SET assigns the value of <expression> to <label>. The difference between SET and EQU is that a label defined by SET may be re-defined later in the program by another SET. A label defined by SET may not be defined by EQU, nor may it be used in the label field of a line of code.

The rules for <expression> are the same as for EQU.

DC (DEFINE CONSTANT)

Format:

```
DC.B  <expression> [,<expression>...]

DC.W  <expression> [,<expression>...]

DC.L  <expression> [,<expression>...]
```

The DC pseudo-ops generate bytes, words or long-words of code whose value is equal to <expression>. More than one constant may be defined by a single DC statement, by using a list of expressions.

Examples:

```
DC.B  'hi'

DC.B  0,0,1,-5,$A9,$FF

DC.W  $4504,7000,-1,'WD'

DC.L  $1801A4C9,'LONG'
```

NOTE: The DC.B pseudo-op may cause the IPC to end up on an odd address after assembling an odd number of bytes. This will cause instructions following the DC.B to assemble on odd addresses, which is illegal. See DS.L below for a way around this problem.

If the expression does not evaluate to a number within the size range specified (8, 16 or 32 bits), an "S" error occurs.

When entering an ASCII string in a DC.W or DC.L statement, the string packed into memory as one character per byte. If an odd number of bytes is entered, the last word (or long word) is zero-filled on the right, to the nearest word (or long word) boundary.

Examples:

```
DC.W    'A'     would put  'A',Ø  in memory

DC.W    'ABC'   would put  'A','B','C',Ø  in memory

DC.L    'A'     would put  'A',Ø,Ø,Ø  in memory

DC.L    'AB'    would put  'A','B',Ø,Ø  in memory
```

DS (DEFINE STORAGE)

Format:

```
DS.B  <expression>

DS.W  <expression>

DS.L  <expression>
```

The DS pseudo-ops reserve space by advancing the IPC past a number of bytes, words or long words. <expression> determines the number of bytes, words, or long words skipped. No code is generated by DS pseudo-ops.

Examples:

```
DS.B  14

DS.L  1
```

NOTE: The DS.B and DC.B pseudo-ops may cause an odd number of bytes to be assembled. This causes the IPC to end up on an odd address, and subsequent instructions will be assembled on odd addresses, which is illegal. To avoid this problem, always follow DC.B and DS.B pseudo-ops with a "DS.L 0" line. This insures that no machine instructions will begin on odd addresses. DS.L used with an argument of zero bumps the IPC to the nearest even boundary.

Example:

```
DS.B  9    Reserve some space, then
DS.L  0      align IPC to a word boundary.
```

END

Format:

```
END  [<addr>]
```

The END statement tells the assembler that no more lines of source code exist in the file. Each file must end with an END, or the assembler will attempt to read past the end of the file and an error will occur.

The last (or only) source file in a program may have an address, <addr>, in the argument field of its END statement. <addr> tells DOS where to begin execution of the program, if and when the program is executed as a .SYS file. If <addr> is omitted, the assembler assumes the program begins at the beginning, and will set <addr> by default to the first instruction in the program.

PAGE

Format:

```
PAGE
```

The PAGE pseudo-op tells the assembler to output a form-feed during the assembly listing. It may be used to break up a listing into convenient pieces. PAGE has no effect on the object code.

LLEN

Format:

```
LLEN  <length>
```

LLEN instructs the assembler that it may print lines containing up to <length> characters. The default value for <length> is 85, corresponding to the width of the CGC Overlay screen. <length> may be set to suit the width of your printer, and must be a number between 0 and 255.

Example:

```
LLEN  132
```

NOLST

Format:

```
NOLST
```

NOLST turns off the assembly listing. It is typically used in conjunction with LIST, to cause only selected parts of the program to be listed during assembly.

LIST

Format:

```
LIST
```

LIST resumes the assembly listing, following a NOLST. LIST is in effect by default, unless a NOLST has been encountered.

ADDRESSING MODES

The MC68000 provides a large number of effective address modes, which define the source and/or destination operands in an instruction. This section discusses and provides examples of each of the addressing modes supported by the MC68000, and the proper syntax for their use.

REGISTER DIRECT

Format:

An or Dn where n is a number, 0 thru 7.

This mode operates directly on the contents of a register, D0 thru D7 or A0 thru A7.

Examples:

MOVE.L	A1,A4	Move the contents of register A1 into register A4 (32 bits).
ADD.W	D0,D3	Add the lower 16 bits of the contents of D0 to the lower 16 bits of the contents of D3. Put the result into the lower 16 bits of D3.

ADDRESS REGISTER INDIRECT

Format:

(An) where n is a number, 0 thru 7.

This mode operates on the memory location whose address is in the specified address register, An.

Examples:

MOVE.B	(A0),D1	Move the byte whose address is in A0, into register D1.
ADD.L	D3,(A5)	Move the contents of register D3 (32 bits) into the long word whose address is in A5.

ADDRESS REGISTER INDIRECT WITH POSTINCREMENT

Format:

(An)+ where n is a number, 0 thru 7.

The operand is pointed to by the specified address register, as it was in Address Register Indirect. After the address is used, the register An is incremented by 1, 2, or 4, depending on whether the operation specifies byte, word, or long word.

Examples:

MOVE.B	D4,(A2)+	Move the low byte of D4 to the address in A2, then add 1 to A2.
ADD.L	(A3)+,D0	Add the long word whose address is in A3, to D0; then add 4 to A3.

ADDRESS REGISTER INDIRECT WITH PREDECREMENT

Format:

-(An) where n is a number, 0 thru 7.

The operand is pointed to by the specified address register, An. Before the address is used, it is decremented by 1, 2, or 4, depending on whether the operation specifies byte, word, or long word.

Examples:

MOVE.L	D2,-(A1)	Decrement A1 by 4; then move 32-bit data from D2 to the location whose address is in A1.
CLR.B	-(A5)	Decrement A5 by 1, then clear the byte at that address (set it to zero).

ADDRESS REGISTER INDIRECT WITH DISPLACEMENT

Format:

d(An) where n is a number, 0 thru 7;
d is a 16-bit displacement.

The displacement d is added to the contents ot register An. This sum provides the address of the operand. d is a used as a sign-extended 16-bit number.

Examples:

MOVE.B	D3,TABLE(A3)	Move the low byte of D3 to the address pointed to by the sum of TABLE plus the contents of A3.
CMP.W	2(A0),D7	Compare the lower word (16 bits) of D7, to the word whose address is two plus the contents of A0. This is the address of the first word after (A0).

ADDRESS REGISTER INDIRECT WITH INDEX

Format:

d(An,Rn.W) where n is a number, 0 thru 7;
d is an 8-bit displacement.

d(An,Rn.L)

The index register (Rn) may be any address or data register. The low word or the entire register may be used as index, depending on whether the suffix ".W" or ".L" is appended to the index register name.

The displacement d is used as an 8-bit sign-extended number. If the low word of Rn is specified, its value is also sign-extended.

Examples:

MOVE.L	Name(A1,D4.W),D5	Move 32-bit data from the address given by: the contents of A1, plus the 16-bit contents of D4, plus the 8-bit value of Name. Data moves into D5.
AND.B	D0,1(A0,D7.L)	Logical AND the low byte of D0, with the byte whose address is given by: A0 plus D7 (32 bits) plus 1. The result is placed in this same memory address.

ABSOLUTE SHORT

The absolute address of the operand is specified. It is used as a sign-extended 16-bit number, which limits this mode to addressing memory between $0000 and $7FFF, and between $FF8000 and $FFFFFF.

> NOTE: If the statement ORG.L is included in the program, this addressing mode will not be used by the assembler. Absolute Long will be substituted (see below).

Example:

```
JMP      $400C      Jump to address 400C (hex)

MOVE.L   A1,$120    Move the 32-bit contents of A1
                    to address 120 (hex)
```

ABSOLUTE LONG

The absolute address of the operand is specified, and is used as a 32-bit number.

Example:

```
ADD.W    $1A2BC,D3  Add the 16-bit data from address
                    1A2BC to register D3, leave the
                    result in D3.

JMP      $800008    Jump to address 800008 (hex)
```

PC WITH DISPLACEMENT

Format:

*+d where d is a 16-bit expression.

*-d

The address of the operand is given by adding the sign-extended displacement d to the current value of the program counter.

The expression "*" has the value of the SECOND word of the code which is generated for this instruction. Therefore, if the instruction has a label NAME, the value of "*" is NAME+2.

Examples:

```
BRA     *+$18   Branch to the address 20 hex bytes
                (16 words) past this instruction.

LEA     *+LABEL-(IPC+2),A0
```

The second example above would be used to generate position-independent code for loading the address of LABEL into A0. It would be operationally similar to the statement

```
MOVEA.L #LABEL,A0
```

which would also load A0 with the address of LABEL. However, the LEA statement above takes fewer bytes, and is preferred if LABEL is within the 16-bit displacement range required by this addressing mode.

PC WITH INDEX

Format:

*+d(Rn.W) where d is an 8-bit expression,
Rn is an address or data register.
*-d(Rn.L)

The address of the operand is given by adding the sign-extended displacement d, to the 16 or 32-bit contents of register Rn, and to the current value of the program counter.

Examples:

```
JMP      *+OFFSET(D3.L)          Jump to the address given
                                 by OFFSET plus the 32-bit
                                 contents of D3, plus the
                                 current Program Counter.

MOVE.W   *+(DATA-(IPC+2))(A0.W),D4

                                 Move a word into D4
                                 from the address which is
                                 beyond label DATA by an
                                 amount contained in A0.W
                                 (the low 16 bits of A0).
```

IMMEDIATE

Format:

#<expression>

Immediate data is always preceded by the pound sign (#). It is used to specify an absolute number other than absolute addresses. The # character tells the assembler to use immediate data, rather than an address. Consider the distinction between these two statements:

```
MOVE.L  $400,D0     Move a long word from address $400
                    into D0 (getting data from memory
                    at address 400 hex).

MOVE.L  #$400,D0    Move the hex number $400 into D0.
```

Certain instructions require immediate data; for example, Move Quick:

```
MOVEQ.L #10,D1      Put decimal 10 into D1.
```

ERRORS

The assembler indicates errors as a one-character abbreviation, to the left of the source line listing. Errors are always printed regardless of the NOLST pseudo-opcode or the -L (suppress listing) option.

Error Code	Description
M	Mode error (wrong operand type)
Z	Size error (wrong size operand)
C	Code error (unrecognized opcode)
Ø	Division by zero
D	Doubly defined label
U	Undefined label
A	Argument error in operand
(	Parentheses error
L	Label too long
E	Expression error
#	Error in number
O	Overflow (number out of range)

The following are non-fatal errors:

~	Phasing error (label does not agree with IPC)
S	Storage Error (overflow from a DC instruction)

Phasing errors are usually the result of other errors earlier in the program, in which a statement did not assemble properly. It is a good practice to fix other errors first, and phasing errors will usually disappear as a result.

APPENDIX A - PROGRAMMING THE CGC 7900

APPENDIX A - PROGRAMMING THE CGC 7900

This section deals with programming techniques applicable to the Chromatics CGC 7900. Two important concepts are discussed, which greatly simplify the process of adding user-written software to the CGC 7900: modules, and jump tables.

Modules are sub-programs which are loaded into memory (RAM or EPROM). When the system is booted, all modules are "linked" together in a fashion which allows any of them to be executed by ASCII code sequences. All of the features implemented in the 7900 are written as modules: this includes I/O drivers, Plot submodes, Mode, Escape and User codes. Using the information in this section, the user can write modules which perform custom functions, and link them into the CGC 7900 system.

Jump tables provide access to important routines in the CGC 7900 software. Functions such as character input and output, Escape code processing, plotting primitives (dot and vector), and the programmable sound generator, can all be accessed thru the jump tables described in this section. Judicious use of these tables prevents the user from "re-inventing the wheel" in his own programs.

MODULES

A module is a sub-program, written according to a list of guidelines so that it may be linked into the CGC 7900 software. There are seven types of modules:

B: Boot only. This module contains code which is executed at boot time, but not otherwise used by the system.

I: Input device. This is a driver which interfaces a physical input device to the system. Currently defined "I" modules drive the keyboard and serial port.

O: Output device. This is a driver which interfaces a physical output device to the system. Currently defined "O" modules drive the serial ports, windows, and keyboard lights.

Mode: Mode code. This module performs functions which modify the attributes of a window. The window software calls Mode modules when it receives a Mode code sequence.

Plot: Plot submode. This module describes a Plot submode, and is called when a window receives a Plot code sequence.

Escape: Escape code. This module is called by the Escape code processor when an Escape code sequence is received. Escape codes generally alter the status of the entire system.

User: User code. This module is called by the Escape code processor when a User code sequence is received. User codes generally alter the status of the entire system, or cause execution of a controlling program (such as DOS).

Modules may exist in EPROM or in RAM. Most of the EPROM firmware in the 7900 consists of modules. For each Mode, Plot, Escape and User code sequence recognized by the 7900, there is a module in firmware which defines the actions taken by that code sequence. All modules in the system are "linked" whenever: the system is powered-up; the RESET key is pressed; or the keys CTRL and BOOT are pressed together.

The process of "linking" means that the system is scanned for modules, the addresses of all modules are loaded into dispatch tables, and any necessary initialization is performed. Note that this "linking" is done sequentially, through EPROM, then through any RAM modules which may have been loaded by the user. This means that two or more modules may define the same code sequence, and the LAST one linked will be the one in the dispatch table after linking is complete. This makes it easy for a user module to re-define a system function, with his module replacing the firmware module which has the same identifying code at its beginning.

Mode, Plot, Escape and User modules may accept arguments. The arguments required by the module are defined by the module, and the system automatically parses arguments before passing control to the module. This relieves the user from writing argument parsing routines, and insures that all arguments are parsed in a consistent manner.

A module must save any registers it modifies, and restore the registers before exiting. As decribed below, several registers are pre-loaded before the module is executed. These registers provide the module with system status information.

THE LINKING PROCESS

The system scans two areas for modules when linking is performed: first, each EPROM pair on the 7900 Raster Processor Board is checked. If an EPROM pair is installed, and if a valid module is found at the start of the EPROM, linking proceeds thru the EPROM. Linking terminates when an invalid length descriptor or an invalid module type code (one not in the set B, I, O, etc.) is found. Linking then proceeds to the start of the next EPROM pair.

After all EPROM modules have been linked, the system checks for RAM modules. RAM modules, if they exist, must be loaded into system RAM at the address pointed to by MDLE. (MDLE is a pointer in CMOS memory, and its contents may be altered with the "Thaw" command. See the 7900 User's Manual for details.) The default address for RAM modules is $1F000; this allows 4K of space for modules, with a single Buffer Memory card installed.

If RAM modules exist, they must follow all the rules described in this section, with an additional provision: to indicate the presence of RAM modules, the bytes 'MDLE' must be loaded into RAM at the start of the first RAM module:

```
ORG.L   $1F000      Org where Thaw wants us to be
DC.L    'MDLE'      Identify we have RAM modules here

etc.
```

The bytes 'MDLE' must NOT be included if a module is being put into EPROM.

Modules are expected to be "back to back", existing in consecutive words of code through memory. This requires that any tables or other data used by a module must be WITHIN the module, not after it.

The last module should end with the following statement:

```
DC.L    -1,-1
```

This indicates to the linker that no more modules follow.

MODULE CONSTRUCTION

Each module begins with a length descriptor. The length is used to determine where one module ends, and where the next one begins. The address of each module is determined at boot time during the linking process, and loaded into a dispatch table for future reference.

Next, a module contains one of the characters B, I, O, Mode (Control-A), Plot (Control-B), Escape (Control-[), or User (Control-U), to define the type of module. This is immediately followed by a character which uniquely identifies that module.

The remainder of the module is variable, depending on the type of module you are writing. Each of the seven types of modules is discussed in this section, and examples are provided.

BOOT MODULES

A Boot module is executed upon power-up, when the system RESET key is pressed, or when the system is booted (by pressing the CTRL and BOOT keys). A Boot module might be written to initialize a piece of hardware, or to pre-load the Case Table for interfacing to a certain host computer.

The Boot module contains a length descriptor (word), the character 'B' followed by a dummy character, and the code to be executed. It ends with a RTS instruction.

```
          ORG.L   $1F000        Org where Thaw says to Org
          DC.L    'MDLE'        Required for RAM modules

          DC.W    MdleEnd-IPC   This is our length
          DC.B    'B',0         Identify a Boot module

*
*     Boot code begins here...
*        .
*        .
*     and ends here.
*

          RTS

MdleEnd   EQU     IPC
          DC.L    -1,-1         Make sure linking ends here
          END
```

INPUT/OUTPUT MODULES

I/O modules define the interface between a physical device, such as a printer, and the logical device assignment structure in the 7900. The I or O module must be responsible for handling all transfers to or from the device, checking the status of the device (if applicable), and booting the device (if necessary). Note that the Boot section of an I or O module performs the function of a B module.

The I or O module begins with a length descriptor (word). This is followed by the character 'I' or 'O', defining an input or output module, and a character between A and Z to identify this particular I/O module. This character (A-Z) is used in the "Assign" command to identify the phyical device which is being assigned.

Two words of code must follow the identifying character. These must be either SHORT branches to Boot and Status sections of the module, or RTS instructions. The Status section is used in an 'I' module to see whether a character is ready to be read. The Status code should return the Z flag SET if no character is available, or clear it if a character is available. If the Status code returns Z set, the system will not execute the main code.

The Status section is not used in an 'O' module. An 'O' module must not return until it has completed processing a character.

The main body of the module follows, terminated by a RTS. The Boot and Status portions of the module must also terminate with a RTS. The system passes a character to an Output module in register D0, and expects an Input module to return a character in D0. The low 8 bits of the register are used.

Note that a single device capable of both input and output requires two modules, one I and one O. (Our device 'Q' below may have an I module and an O module associated with it.)

```
*
*     Sample Input module (type 'I') for a device named Q.
*
          ORG.L    $1F000       Org where Thaw says to Org
          DC.L     'MDLE'       Required for RAM modules

          DC.W     MdleEnd-IPC  Our length
          DC.B     'I','Q'      Input module for device 'Q'

          BRA.S    Qboot
          BRA.S    Qstat

*
*     Main code for inputting a character from device Q.
*

          MOVE.B   QDataPort,D0   Get input data
          RTS


*
*     Code for booting device Q.  (Executed at Boot time)
*

Qboot     MOVE.B   #0,QCtrlPort     Initialize device Q with
          MOVE.B   #$FF,QCtrlPort   funny little numbers.
          RTS

*
*     Code for checking status of Q.
*

Qstat     MOVEM.L  D0,-(SP)         Save the register we use
          MOVE.W   QStatusPort,D0   Get 16-bit status word
          BTST.L   #3,D0            Bit 3 says ready if SET
          MOVEM.L  (SP)+,D0         Restore D0 (MOVEM doesn't
                                    change any flags)
          RTS

MdleEnd   EQU      IPC
          DC.L     -1,-1            No more modules here
          END
```

ARGUMENT PARSING

The module types discussed beyond this point may have arguments associated with them. An argument is a set of characters or numbers which is passed to the module. Mode, Plot, Escape and User modules may accept arguments. The arguments required by a module are defined by that module, and are parsed by the system before the module is called. The module then simply picks up its arguments and processes them.

The system will parse ten types of arguments:

Arg type	Description
1	A single character
2	A string of characters, delimited by a space, comma, or semicolon
3	A string of characters, delimited by a semicolon ONLY
4	A signed 16-bit decimal number (or a number in Binary Coordinate form, if the system is in Binary Mode)
5	A 16-bit hexadecimal number
6	A decimal number, or the X component of a coordinate defined by the cursor
7	A decimal number, or the Y component of a coordinate defined by the cursor
8	An X-Y coordinate pair (combination of 6 and 7, will accept cursor position)
9	Any number of coordinate pairs, delimited by a semicolon
A	A decimal number, scaled but without translation

Argument types 6, 7, 8 and 9 are designed to parse coordinate data. Each of these will scale and translate arguments according to the scale factors in use in the window which executed the module.

MODE MODULES

A Mode module is executed when a window receives the Mode code sequence identifying that module. Mode modules are expected to affect only the window which called them, although nothing in the system will prevent a Mode module from affecting other windows or other aspects of the system.

A Mode module consists of a length descriptor (word), followed by the Mode character (control-A, decimal 1), and a character which uniquely identifies the module. This character may be any ASCII character above '0' (hex $30).

The next long word in the Mode module defines a list of arguments, to be parsed and passed to the module. Each nybble (4 bits) of the long word specify one of the ten argument types listed above. The argument list in this long word is right-justified, and the least-significant nybble defines the FIRST argument to be parsed. The long word must be left-filled with zeros to indicate the end of the argument list. Up to 8 arguments may be defined in this long word.

Example:

```
DC.W    MdleEnd-IPC  Length Descriptor
DC.L    $00000144    Argument list
```

This list would specify that the module requires two decimal numbers, followed by a single character.

Arguments to a Mode module are put onto the "A1 stack." That is, the Mode module may assume that its arguments are waiting for it at the address pointed to by (A1), and successive bytes.

```
          ORG.L   $1F000        Org where Thaw says to Org
          DC.L    'MDLE'        Required for RAM modules

          DC.W    MdleEnd-IPC   Length descriptor

          DC.B    Mode,'Z'      Module identifier
Mode      EQU     1             (Mode is Control-A)

          DC.W    $0000         Our arg list (one decimal #
          DC.W    $0014         and one character)

*
*     Code for module "Mode Z" begins here.
*
*     The user would type Mode Z <n>, <c>
*
*     where <n> is a decimal number (the terminating
*               comma is required by the arg parser)
*
*           <c> is a single character
*

          MOVEM.L D0-D2,-(SP)   Save what we use

          MOVE.W  (A1)+,D2      Get first argument (decimal #)
          MOVE.B  (A1),D0       Get second argument (char)

*
*     Now do something useful with the args...
*     Print character <c> on the screen, <n> times.
*

          CLR.L   D1            Specify Logical Output Dev 0
PRNT      JSR     CHAROUT       Print the char in D0
CHAROUT   EQU     $800008       (See Jump Table description)

          SUBQ.W  #1,D2         Decrement the count
          BNE.S   PRNT          Do until D2 = 0

          MOVEM.L (SP)+,D0-D2   Restore what we saved
          RTS

MdleEnd   EQU     IPC
          DC.L    -1,-1         End of modules
          END
```

PLOT MODULES

A Plot module performs a graphics function. All of the plotting features in the 7900 firmware are written as Plot modules. Plot modules generally accept coordinate data as arguments, and perform some plotting function based on this data. The "linking" procedure used for all modules means that you can write your own Plot modules, link them in to replace existing modules in the 7900 firmware, and implement any features you desire.

Plot modules begin with a length descriptor (word). This is followed by the Plot character (control-B, decimal 2) and a character which identifies the module. The identifier may be any character above ASCII "@" (hex $40). The next two words specify arguments to be passed to the Plot module. This argument list is similar, but not identical, to the argument list for Mode modules.

Once a Plot code sequence has been entered, execution of the Plot module begins. The module is then repeatedly called, each time enough arguments have been entered to satisfy its list. The FIRST time a Plot module is entered, it may wish to perform a different sequence of instructions to initialize itself. A status bit tells the module whether it is being entered for the first time, or on subsequent calls. Also, the second word of the argument list is used to select arguments for the first call, and the first word selects arguments for subsequent calls:

```
DC.W    ModEnd-IPC
DC.B    Plot, 'Z'

DC.W    $0011       Two chars for repeated calls
DC.W    $0008       Coord arg for first call
```

In this example, the Plot module requires a coordinate argument when it is first executed. After the coordinate argument is satisfied, the module is entered. The system will, from that point on, accept two single characters before entering the module. (This is the argument scheme used by Incremental Vector.) The process of scanning for a pair of characters, and calling the module, continues until another Plot submode is entered or until the user turns "Plot Off."

On entry to the Plot module (and other modules), certain registers are set up for convenience. Applicable to this discussion is D7, which contains window status. The "Submode" bit of D7 (bit 17) is set when a Plot module is first entered, to indicate that "this Plot Submode was just entered." If a Plot module cares about whether the submode has just been entered, it must check this bit and perform any necessary initialization if the bit is set. Then, it must clear the "Submode" bit.

If the Plot module does not make this distinction, and does not care whether it has just been entered, then the argument list should be duplicated in the first and second word:

```
DC.W    ModEnd-IPC
DC.B    Plot,'Z'

DC.W    $0008          A coord for repeated calls
DC.W    $0008          and also for the first call.
```

Note that Plot modules may only parse FOUR arguments before execution begins; Mode modules could have up to eight.

So that Plot code and Mode code arguments will not conflict, Plot arguments are stacked on the "A3 Stack" and may be picked up from the addresses pointed to by A3.

Remember that more than one window at a time may be in the same Plot Submode. A Plot module should not store any local data which could interfere with another window calling the Plot module. Local data should be stored in the window table, or somehow be localized to the window.

```
*
*    A sample Plot module.
*
         ORG.L    $1F000
         DC.L     'MDLE'

         DC.W     ModEnd-IPC
         DC.B     Plot,'Z'
Plot     EQU      2              (Plot is Control-B)

         DC.W     $0008          One coord pair normally,
         DC.W     $0088          Two pair the first time thru.

Submode  EQU      17
         BTST     #Submode,D7    Did we just enter the submode?
         BEQ.S    Norm           No... normal entry
*
*    Submode was just entered.  We can initialize ourself
*    in this block of code if necessary.
*
         BCLR     #Submode,D7    Prepare for next entry
         RTS


*
*    Come here if submode was NOT just entered.
*

Norm     MOVEM.L D0-D2,-(SP)     Save registers
         MOVE.W   (A3)+,D0       Get X argument into D0
         MOVE.W   (A3),D1        Get Y argument into D1

*
*    Now, do something with the arguments here....
*    Maybe plot a vector or something.
*
         MOVEM.L (SP)+,D0-D2     Restore registers
         RTS
```

ESCAPE AND USER MODULES

Escape and User modules are identical to each other, and are similar to the Mode modules discussed earlier.

An Escape or User module begins with a length descriptor (word), followed by the Esc ($1B hex) or User ($15 hex) character, and a single ASCII character which identifies the module. The identifier may be any character above ASCII '@' ($40 hex).

The next two words define an argument list, exactly like a Mode module. All arguments (up to eight) are parsed by the Escape Code Processor and are passed to the Escape or User module. Arguments are found on the A1 stack (pointed to by A1). This does not conflict with the stack of Mode arguments because Escape and User codes are processed by a different routine than Mode codes.

Remember that Escape and User codes have identical priority in the 7900 code processing scheme. Escape and User modules are not specific to a window, so they should be used to implement functions affecting the entire machine.

```
*
*    A sample Escape code module to play with the
*    lights on the keyboard.  This could also have
*    been written as a User module.
*
         ORG.L    $1F000
         DC.L     'MDLE'          Identify a module is here

         DC.W     MdleEnd-IPC     Length descriptor
         DC.B     Esc,'X'         Escape X is our sequence
Esc      EQU      $1B

         DC.W     $0000
         DC.W     $0004           We want one decimal #

         MOVE.W   (A1),Keybrd     Send it to the keyboard
Keybrd   EQU      $FF8080         (Keyboard address)

         RTS

MdleEnd  EQU      IPC
         DC.L     -1,-1

         END
```

REGISTER SETUP FOR MODULES

When a module is executed, several registers are pre-loaded for convenience. The following table defines what each register is used for, when each type of module is entered.

Module	Register Usage
B (boot)	No registers pre-loaded.
I (input)	No registers pre-loaded.
O (output)	No registers pre-loaded.
Mode, Plot	A0: Points to base of Window Table for the window which called the module (see Window Table description). A1: Points to Mode arguments (parsed before module execution begins). A3: Points to Plot arguments (parsed before module execution begins). D7: Contains window status for the window which called the module (see Window Table description).
Escape, User	A0: Points to base of the Window Table for window A (the Master Window). A1: Points to arguments (parsed before module execution begins). D7: Contains window status for window A and Escape code status.

WINDOW TABLE

512 bytes of data are allocated as a Window Table for each active window. These bytes hold the current status of the window, including such items as color, blink, scale, window limits, and most other window attributes. Window attributes are usually set by Mode code sequences, and Mode modules will often want to alter items in a Window Table.

The following chart lists the location of each item in the window table, by giving an offset into the table where each item may be found. An item can be altered by a module by using the listed offset as a displacement from (A0), since A0 is pre-loaded with the base of the Window Table. For example,

```
FrgCol  EQU     $8E             Offset in table
                                for Foreground color

        MOVE.W  FrgCol(A0),D0   Get color into D0
```

In this table, entries are marked with ".B", ".W", or ".L", to indicate the appropriate data size (where possible).

```
$00 .L    TVALUE:   Temporary storage area (reserved)
$04 .L    Arglst:   Mode argument list
$08 .L    Parglst:  Plot argument list
$0C .L    STATUS:   copy of D7 status long word
$10 .L    ^Argstk:  pointer to Mode argument stack
$14 .L    Argdsp:   Mode dispatch address
$18 .L    ^Prgstk:  pointer to Plot argument stack
$1C .L    Prgdsp:   Plot dispatch address (submode)

      Window variables are stored beginning here
            and occupy one word each.

$20 .W    Window Variable A
$22 .W    Window Variable B
   .
   .
$5C .W    Window Variable _
$5E .W    Window Variable `
```

The following items are also Window Variables but are used for system data as well.

$60	.W	AcursX:	Overlay cursor X position
$62	.W	AcursY:	Overlay cursor Y position
$64	.W	CursX:	Bitmap cursor X position
$66	.W	CursY:	Bitmap cursor Y position
$68	.W	AwindX0:	Overlay window upper left corner X
$6A	.W	AwindY0:	Overlay window upper left corner Y
$6C	.W	AwindX1:	Overlay window lower right corner X
$6E	.W	AwindY1:	Overlay window lower right corner Y
$70	.W	WindX0:	Bitmap window upper left corner X
$72	.W	WindY0:	Bitmap window upper left corner Y
$74	.W	WindX1:	Bitmap window lower right corner X
$76	.W	WindY1:	Bitmap window lower right corner Y
$78	.W	CharXZ:	Character X raster size
$7A	.W	CharYZ:	Character Y raster size
$7C	.W	CharDX:	Character delta X after write
$7E	.W	CharDY:	Character delta Y after write
$80	.W	CharXM:	Character X multiplier
$82	.W	CharYM:	Character Y multiplier
$84	.W	XVmin:	Virtual X minimum value
$86	.W	YVmin:	Virtual Y minimum value
$88	.W	Tabcol:	Tab stop spacing
$8A	.W	Vecwid:	Vector width
$8C	.W	BkgCol:	Background color
$8E	.W	FrgCol:	Foreground color
$90	.W	PlaneE:	Planes enabled
$92	.W	CursCol:	Cursor color (not used, reserved)
$94	.W	OldX:	Rubber band X position
$96	.W	OldY:	Rubber band Y position
$98	.W	XScl:	Virtual X Scale value
$9A	.W	YScl:	Virtual Y Scale value
$9C	.L	Charadr:	Character set base address
$A0	.L	Endbuf:	End of arguments for virtual coordinate
$A4	.L	Plotdot:	Dispatch address for dot plotting
$A8	.L	Plotvect:	Dispatch address for vectors

The following five items are used for raster processor operations in the window.

$AC	.W	WXsrc:	X source raster operating point
$AE	.W	WYsrc:	Y source raster operating point
$B0	.W	WDXsrc:	Delta X for source raster
$B2	.W	WDYsrc:	Delta Y for source raster
$B4	.W	Wctrl:	Control bytes for rasters

$B6-$F5	Curstg:	32 words for cursor pixel storage
$F6-$135	Argstk:	Argument stack for Mode args (32 words)
$136-$1B5	Pargstk:	Argument stack for Plot args (64 words)

The following four bytes contain the current Overlay color, blink and transparency attributes.

$1B6	AentSt:	Transparency
$1B7	AentBC:	Background color
$1B8	AentFC:	Foreground color
$1B9	AentCh:	ASCII character portion

$1BA-$1FF	Reserved for future expansion

WINDOW STATUS AND ESCAPE CODE STATUS

As shown before, register D7 is pre-loaded with status information when certain types of modules are executed. The bits in D7 are defined as follows:

Bit		Meaning
0	modeF:	Mode flag used by window processor
1	plotF:	Plot flag used by window processor
2	Moredat:	Control bit used by argument scanners
3	Negnum:	Control bit used by argument scanners
4	visctrl:	SET when visible ctrls are on
5	Pltmode:	SET when in a plot submode (not alpha)
6	Overlay:	SET when Overlay on (not Bitmap)
7	Curson:	SET when cursor is on
8	Fillon:	SET when fill is on
9	Blinkon:	SET when blink is on
10	Rollon:	SET when roll is on
11	OvrStrk:	SET when overstrike is on
12	Binmode:	SET when binary mode is on
15	Rubron:	SET when rubber band is on
16	Patton:	SET when patterns are on
17	Submode:	SET when plot submode just entered
18	Cursin:	SET when Bitmap cursor in screen RAM
19	BinTwo:	Flag for binary coordinate parser
20	VScale:	SET when scaling is on
21	A7on:	SET when A7 character set active
23	Local:	SET in LOCAL mode
24	Full:	SET in FULL duplex
25	Escflg:	Escape code processor flag
26	Usrflg:	Escape code processor flag
27	Create:	SET if Create is on
28	LitP0:	Escape code processor flag
29	LitPl:	Escape code processor flag
30	Literal:	SET if Literal Create is on
31	Escdone:	Escape code processor flag

Other bits are reserved.

JUMP TABLES

This section desribes the utilities available in CGC 7900 PROM firmware. Each of these routines may be accessed through a subroutine call (JSR) to the appropriate address. Note that BSR would not work because in general, jump tables are located more than a 16-bit displacement from your program in RAM.

Registers used in each routine are defined in this section. Unless noted, the routine only alters registers as necessary to return values to the caller. The notation "D1.W" means that the low word (16 bits) of D1 are used by the routine, D0.B means the low byte of D0, and so on.

Name: CHAROUT
Address: $800008

Entry: D0.B = character to pass to logical device
D1.W = logical device number (0 to 4 are defined)

CHAROUT is the system character-out routine. It passes a character to a logical output device. The system's device assignment structure will then pass the character on to a physical device, if possible. Logical output device 0 is normally used to put characters on the screen, and device 1 is normally connected to the serial port. If the character is part of a Mode or Plot code sequence, it will be processed when it reaches a Window (physical device). Escape and User codes are NOT processed by CHAROUT, but are treated as normal characters. To process Escape and User codes, see CTRLOUT, CTRLIN and ESCPROC below.

Name: CHARIN
Address: $80000C

Entry: D1.W = logical device number to read from

Exit: Zero flag SET if no character was available
Zero flag CLEAR and character in D0.B if available

CHARIN is the system character-in routine. It reads a character from a logical input device. Reading from device 0 will normally get a character from the keyboard (if available), device 1 will normally be the serial port.

To wait for a character from CHARIN, use a loop on the EQ condition:

```
Loop    JSR     CHARIN    Get a character, if any
        BEQ.S   Loop      No character yet
```

Escape and User codes are not processed by CHARIN, but are treated as normal characters. See CTRLOUT, CTRLIN and ESCPROC below.

Name: CTRLOUT
Address: $800010

Entry: D0.B = character to pass to logical device
D1.W = logical device number
Zero flag must NOT be set

CHAROUT

CTRLOUT is like ~~CTRLIN~~, but processes Escape and User codes before passing the character on to the logical output device. It does this by calling ESCPROC (see below), then calling CHAROUT. Note: do not call CTRLOUT if the zero flag is set. This will cause your character to be ignored.

Name: CTRLIN
Address: $800014

Entry: D1.W = logical device number to read from

Exit: Zero flag SET if no character was available
Zero flag CLEAR and character in D0.B if available

CTRLIN is like CHARIN, but processes Escape and User codes before returning a character to you. It does this by calling CHARIN and then ESCPROC. If an Escape or User code was entered, it will be "eaten" by ESCPROC and the zero flag will be returned to you, indicating that no character was available.

Name: ESCPROC
Address: $800018

Entry: D0.B = character to process
Zero flag must NOT be set

Exit: Zero flag SET if character was eaten by ESCPROC
Zero flag CLEAR if character is still available

ESCPROC handles Escape and User code processing. It is used by CTRLIN and CTRLOUT, and may also be called directly. ESCPROC detects Escape and User codes, and processes them by setting the zero flag to indicate that the character was processed (and thus should not be considered available). After the Escape or User code is detected, ESCPROC will process subsequent characters to satisfy the argument list of that particular Escape or User function, then execute the function. This will normally be transparent to the user.

Name: BOOT
Address: $80004C

BOOT boots the system. It does not return to the caller. One reason for calling BOOT would be to link in any RAM modules you have loaded into system memory.

Name: NOISE
Address: $800054

Entry: A0 points to tone descriptor block (14 bytes)

Exit: A0 is incremented past block

NOISE feeds data to the sound generator. 14 bytes are loaded sequentially into the tone chip. These bytes go into registers 0 through 13 of the tone chip (a General Instruments AY-3-8910), and control the following attributes:

Register #	Purpose	
0	Fine Tune A	(8 bits)
1	Coarse Tune A	(4 bits)
2	Fine Tune B	(8 bits)
3	Coarse Tune B	(4 bits)
4	Fine Tune C	(8 bits)
5	Coarse Tune C	(4 bits)
6	Noise Period	(5 bits)
7	Output Enable	
8	A Amplitude	(5 bits)
9	B Amplitude	(5 bits)
10	C Amplitude	(5 bits)
11	Envelope Period Fine	(8 bits)
12	Envelope Period Coarse	(8 bits)
13	Envelope Shape/Cycle Control	(4 bits)

The tone generator has three voices, A, B, and C, each of which can be programmed to produce tone or noise. If a given voice is programmed for both tone and noise, noise will usually dominate. Tone and/or noise are enabled by register 7:

7	6	5	4	3	2	1	0
X	X	An	Bn	Cn	At	Bt	Ct

A zero on any of the "n" bits enables noise from that channel, and a zero on any of the "t" bits enables tone from that channel. Unused channels are turned off by writing ones in the desired bits.

Registers 8, 9 and 10 control the output amplitudes:

7	6	5	4	3	2	1	0
X	X	X	A	manual level ctrl			

A one in bit 4 specifies the channel's amplitude to be controlled by the envelope generator (Auto mode). If bit 4 is a zero, the amplitude is fixed by the value in bits 0-3.

The envelope generator is controlled by register 13:

7	6	5	4	3	2	1	0
X	X	X	X	cont	attk	alt	hold

Bits 0-3 describe the envelope with "continue," "attack," "alternate," and "hold." See General Instruments literature for the envelope waveforms.

Name: PRTDEC
Address: $800058

Entry: D0.W = decimal number to convert to ASCII
A1.L = pointer to buffer where ASCII goes

PRTDEC prints a decimal number as an ASCII string. The string is placed into memory at (A1)+.

Name: SIND0
Address: $800064

Entry: D0.W = angle in integer degrees

Exit: D0.W = sine of that angle

SIND0 takes the sine of an angle and returns the value as a 14-bit fraction. The form of the fraction is:

S is the sign of the value (1 is negative), M is the mantissa (zero except if the value is one or negative), F are the fractional bits. The following example uses SIND0 to compute Y * SIN(theta). D0 is the angle theta, and Y is in the lower word of D1. The value is returned in D1.

```
JSR     SIND0   Get sine of theta
MULS    D0,D1   Y=Y*SIN(theta)
ASL.L   #2,D1   Adjust for 14-bit fraction
SWAP    D1
EXT.L   D1      Clear garbage from hi word
```

Name: PLRRCT
Address: $800068

Entry: D0.W = radius
D1.W = angle in integer degrees

Exit: D0.W = X
D1.W = Y

PLRRCT performs polar to rectangular conversion, using SIND0 and a technique similar to the example above.

ame: READJOY
ddress: $800070

ntry: A1.L = pointer to joystick X, Y or Z axis

xit: D0.W = value read from joystick

EADJOY returns the current value of a joystick axis as a 10-bit umber in the range 0 to 1023. A1 must be set to the address of ne of the joystick axes, as follows:

X $FF80C6
Y $FF80CA
Z $FF80CC

PLOTTING FUNCTIONS

Many of the plotting primitives in the 7900 may be accessed through jump tables. Plot functions are specific to a window, which means the system must know which window to use for executing the plot routine. Data from the Window Table determines such things as the color of the plotted figure, and whether or not the figure will be filled.

All of the plot routines discussed below require that register A0 be pointing to the base of the Window Table for the window in which the plot will occur. This will be done automatically if your program is linked as a module, but if you are writing a transient, you must load A0 in your program. If plotting in window 0 is desired, you may set up A0 by this code:

```
BtmGWin EQU     $C40          Pointer to W table base
        MOVE.L  BtmGWin,A0    Get pointer
```

Each Window Table occupies 512 bytes. If D0.L contains the window number (0 through 7), the following code would point A0 to the base of any Window Table:

```
        MOVE.L  BtmGWin,A0    Get pointer
        ASL.L   #9,D0         D0=512*D0
        ADD.L   D0,A0         Add to base
```

Some of the functions described below require an argument list, which is passed on the "A3 stack." The values passed to the routine are pointed to by (A3), and the words following (A3). An area of the Window Table called Pargstk (Plot Argument Stack) is normally used to pass arguments, or your program can use other RAM for this purpose. To load the Pargstk area with four values for a vector, the following code could be used:

```
Pargstk EQU     $136   Offset in W table for plot args

        LEA     Pargstk(A0),A3   Point to Pargstk
        MOVE.W  X1,(A3)
        MOVE.W  Y1,2(A3)   Load XY values on A3 stack
        MOVE.W  X2,4(A3)
        MOVE.W  Y2,6(A3)
        JSR     FVECT      Draw a vector
```

Before calling any of these plot routines, the data to be plotted must be scaled to screen coordinates, between 0 and 1023. Data outside this range will be plotted unpredictably.

ame: PLOTXY
ddress: $80005C

ntry: D0.W = X value
D1.W = Y value
A0.L = pointer to Window Table

PLOTXY plots a single dot in the Overlay or Bitmap, at the XY coordinate specified by D0 and D1. PLOTXY vectors through the Window Table entry called Plotdot, which holds the address of a routine to plot a dot in the Overlay or Bitmap. Plotdot may also hold the address of a routine which plots patterns in the Bitmap, if patterns have been enabled. The address in Plotdot is loaded by any of the "Mode O" or "Mode T" commands. The current foreground color of the window is used unless patterns are active.

Name: FVECT
Address: $800060

Entry: A3.L = pointer to X1, Y1, X2, Y2 (words)
A0.L = pointer to Window Table

FVECT plots a vector in the Overlay or Bitmap, from (X1, Y1) to (X2, Y2). FVECT vectors through the Window Table entry called Plotvect, which holds the address of a routine to plot a vector in the Overlay or Bitmap (see the discussion of Plotdot above). The address in Plotvect is loaded by any of the "Mode O" or "Mode T" commands. The current foreground color of the window is used unless patterns are active, OR UNLESS THE VECTOR IS HORIZONTAL. If Y1=Y2, a special fast vector routine is used which writes part of the vector through proprietary Color Status hardware. The color produced by Color Status is determined by loading the Color Status Foreground latch, a word at address $E40016. To use FVECT properly, you must load the Window Table and the Color Status latch with your desired color.

Name: BVECT
Address: $800074

Entry: A3.L = pointer to X1, Y1, X2, Y2 (words)
A0.L = pointer to Window Table

Name: CIRCLE
Address: $800078

Entry: A3.L = pointer to X, Y, radius (words)
A0.L = pointer to Window Table

Name: ARC
Address: $80007C

Entry: A3.L = pointer to X, Y, radius, start, delta (words)
A0.L = pointer to Window Table

Name: CURVE
Address: $800084

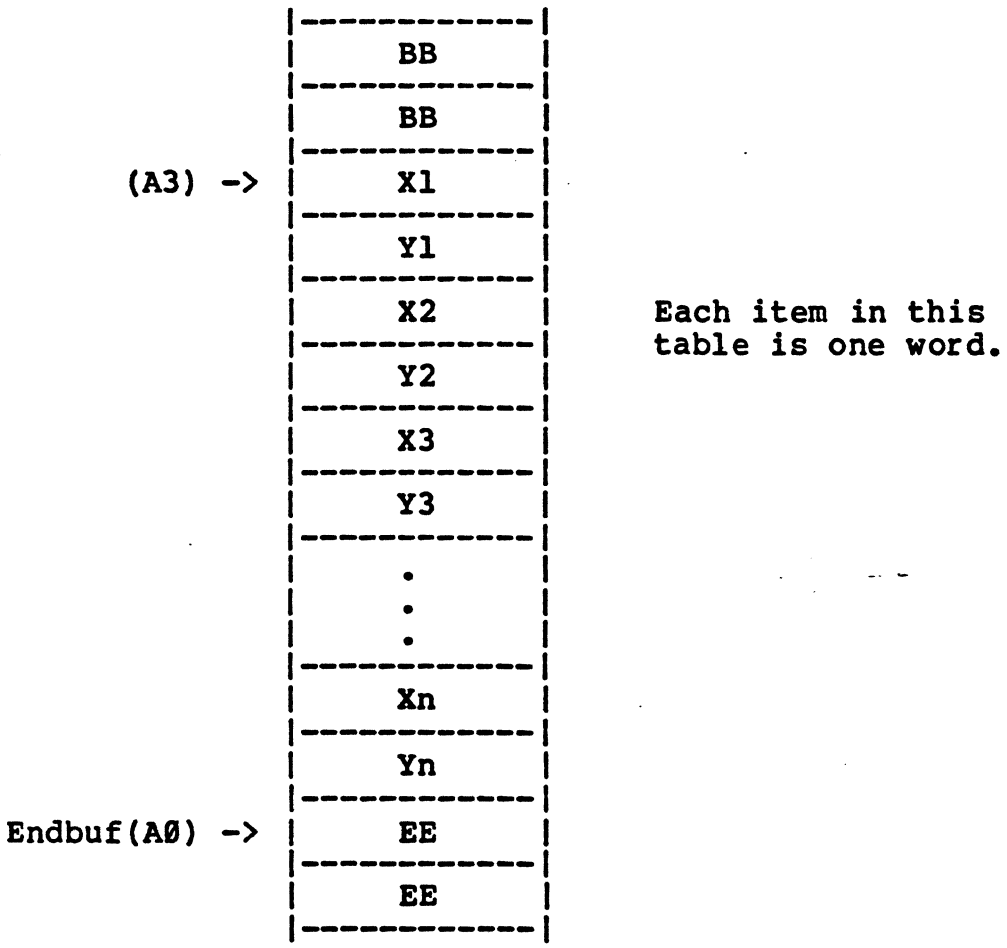
Entry: A3.L = pointer to X1, Y1, X2, Y2, X3, Y3, X4, Y4 (words)
A0.L = pointer to Window Table

Each of these routines plots a figure according to the attributes of the Window Table pointed to by A0. Plotting will occur in the Overlay or Bitmap, with or without patterns, according to the current status of the window.

Name: POLYG
Address: $800080

Entry: A3.L = pointer to coordinate pairs (words)
A0.L = pointer to Window Table

POLYG is similar to the other plot routines described above, except that it can accept a variable-length argument list. The beginning of the list is pointed to by A3.L, and the end of the list is pointed to by an entry in the Window Table named Endbuf. Endbuf is a long word, and it holds the address of the word PAST the last coordinate in the polygon argument list.



Pairs of words before and after the list (marked BB and EE above) are used as scratch areas by POLYG, and your program should allow room for them.

The following code might be used to call POLYG:

```
Pargstk EQU    $136     Offsets in W table
Endbuf  EQU    $A0

        LEA    Pargstk(A0),A3    Point to arg stack
        MOVE.L A3,-(SP)            Save pointer
        MOVE.W X1,(A3)+
        MOVE.W Y1,(A3)+           Load coordinates
        MOVE.W X2,(A3)+             onto A3 stack
        MOVE.W Y2,(A3)+
            .
            .
            .
        MOVE.W Xn,(A3)+           Put last values
        MOVE.W Yn,(A3)+
        MOVE.L A3,Endbuf(A0)      Set up end of list
        MOVE.L (SP)+,A3           Retrieve pointer to
                                    start of list
        JSR    POLYG              Do polygon
```

DOS JUMP TABLES

Name: DOS
Address: $80C008

This is the main entry point to DOS. It requests the user's password and begins accepting DOS commands.

Name: EXDOS
Address: $80C00C

Entry: A1.L = pointer to command line

Exit: A1.L = pointer to next (unprocessed) character in line
D0.B = error code (if any), or zero if no error

EXDOS attempts to execute a transient. The name of the transient should be pointed to (on the command line) by A1. EXDOS calls GETNAM to parse the transient name, OPEN to locate the file on disk, and LOAD to load the file into memory. If successful, execution begins at the transient's start address. The TRANSIENT is responsible for returning A1 and D0 as required above.

Name: OPEN
Address: $80C010

Entry: A0.L = pointer to UFT in use

Exit: D0.B = error code (if any), or zero if no error

OPEN looks up a file on a disk. Before calling OPEN, the UFT (User File Table) should contain the complete filename: primary, secondary, password, and drive. (The UFT is an area of RAM which defines the current status of a file, and includes all of the file's vital statistics. UFT's are discussed later.) See GETNAM for a way to parse the filename. If successful, OPEN will add the following information to the UFT: START, LENGTH, ACCESS, STATUS, BPNTER, BLNGTH, SLOT. If unsuccessful, D0.B holds the error code.

Name: CLOSE
Address: $80C014

Entry: A0.L = pointer to UFT in use

Exit: D0.B = error code (if any), or zero if no error

CLOSE enters a new file into the disk directory. The UFT must be completely built before calling CLOSE. CLOSE is only used on files which have been created by CREATE, not on existing files which have been OPENed. If the file name specified in the UFT already exists, the old file by that name is killed automatically.

Name: CREATE
Address: $80C018

Entry: A0.L = pointer to UFT to be used

Exit: D0.B = error code (if any), or zero if no error

CREATE prepares the largest available free space on the disk for writing. Before calling CREATE, the UFT should contain the complete filename: primary, secondary, password, and drive number. CREATE will add SLOT, START, LENGTH, ORIGIN, ACCESS, BPNTER and BLNGTH. These items will reflect the largest currently available disk space.

Name: LOAD
Address: $80C020

Entry: A0.L = pointer to UFT in use

Exit: D0.B = error code (if any), or zero if no error

LOAD reads an executable file into memory. The file must be in load module form, as a .SYS file. If the file is loaded successfully, RAM location GOADDR ($11B4) will contain the file's normal execution address. LOAD returns to the caller, who may then jump to the address in GOADDR if desired.

Name: RWBYTE
Address: $80C024

Entry: A0.L = pointer to UFT in use

Exit: D0.B = error code (if any), or zero if no error

RWBYTE is the main disk read/write routine. The UFT must contain proper values in BUFFP, MBYTES, DRIVE, CONTROL, BPNTER, BLNGTH, and STATUS. BUFFP is the memory address to/from which data will be moved. MBYTES is the number of bytes. If MBYTES < 128, then data may be transferred to/from an odd memory address. If MBYTES >= 128, data must be transferred to/from even addresses only.

On exit, BUFFP points past the memory location where the last transfer took place. MBYTES will be zero if all requested bytes were transferred, else it will be the number of bytes NOT transferred (due to error). The EOF bit of CONTROL will be set appropriately. BPNTER and BLNGTH will be updated according to the current state of the file.

Name: GETNAM
Address: $80C064

Entry: A0.L = pointer to UFT to use
A1.L = pointer to input buffer (command line)
A2.L = pointer to default filename

Exit: D0.B = error code (if any), or zero if no error
D1.B = last character processed
A1.L = pointer to first unprocessed character

GETNAM parses the input buffer and extracts a file name. All parts of the file name are loaded into the UFT. A string of 11 characters to be used as a default name (if no name was entered) must be pointed to by A2. If no password was entered, the current user password is copied into the UFT. If no drive number was entered, the current drive number is copied into the UFT.

GETNAM returns D1.B with the delimiter it found after the file name. This may be a colon, in which case you must flag to DOS that another command exists on the input line. It may also be a semicolon, used to delimit an option field. A1 should be saved for return to DOS, or used as a pointer to further arguments on the command line (if your program expects any).

Name: PRTMSG
Address: $80C084

Entry: A0.L = pointer to ASCII string (terminated by zero)

PRTMSG transmits an ASCII string to Logical Output Device zero, which will normally display it on the screen. PRTMSG goes through CTRLOUT to allow Escape and User code processing.

Name: PRTHEX
Address: $80C094

Entry: D0.L = data (hex long word)
D1.L = number of hex digits to print (1 to 8)

PRTHEX prints a hex number to Logical Output Device zero (normally the screen). The number is preceded by a dollar sign ($). D1 specifies the number of hex digits to print, and these are taken from the least significant digits of D0. (If D1 = 2, then the low byte of D0 is printed.) The hex number is left-justified, and padded on the right with spaces if necessary, to fill out the number of characters specified by D1. This is the format of hex numbers printed in the disk directory.

Name: GETCLK
Address: $80C098

Entry: No setup required

Exit: D0.L = packed time and date information from clock

GETCLK reads the Real Time Clock and encodes time and date information into a long word. If the clock option is not installed, the long word contains zero.

Name: CLKBCD
Address: $80C0A0

Entry: D0.L = packed time and date in GETCLK format
A0.L = pointer to 19-character buffer

Exit: Buffer is loaded with ASCII time and date

CLKBCD unpacks the time and date information produced by GETCLK. The buffer pointed to by A0 will be loaded with month, day, year, hour, and minute information in ASCII form. A zero byte is appended to the ASCII text so that it can be printed by PRTMSG. If D0 contained zero on entry to CLKBCD, the buffer will be loaded with 18 spaces and a zero.

Name: GETARG
Address: $80C0A8

Entry: A1.L = pointer to input buffer

Exit: A1.L = pointer to character past delimiter
D1.L = hex argument returned
D0.B = zero if no error, non-zero if error

GETARG parses a hex number from an input buffer. The value of the argument is returned in D1. D0 is non-zero if a non-hex character was detected before the delimiter was reached. If D0 is zero, no error was detected. A1 is advanced past the argument.

Name: DOSERR
Address: $80C0BC

Entry: A0.L = pointer to UFT of the file which caused an error
D0.B = error code

DOSERR prints a DOS error message. The drive number of the offending file is printed also. Error codes available in DOSERR are listed in the Appendix. Note that DOS automatically prints error messages if your transient returns to DOS with D0.B non-zero.

INLINE CALLING SEQUENCE

Name: INLINE
Address: $80A00C

Entry: A1.L = pointer to input buffer to be used
D1.W = Logical Input Device number to read from
D7.B = control bits (see below)

Exit: Zero flag SET if the user hit DELETE
Zero flag CLEAR if the user hit RETURN

INLINE is the 7900's general-purpose input routine, used by DOS, the Monitor, and Thaw. It reads a line of up to 83 characters from the user, allowing character editing, Recall Last Line, etc. Bits in D7 control INLINE as follows:

Bit	Meaning if SET
3	Echo the input line to the screen after RETURN is pressed (in expanded form, Modes and tabs executed normally).
2	Process Escape and User codes as they are entered.
1	Use "A7" character set for control-characters displayed in compressed form.
0	Do not display the characters as they are entered (you can't see what you type).

DOS uses D7 equal to $0E, D1 equal to zero, and A1 pointing to the DOS input buffer in low RAM. The input buffer supplied to INLINE should be at least 85 characters long. The end of the user's input line is indicated by a Return character in the buffer.

CMOS MEMORY ALLOCATION

1096 bytes of CMOS or static memory are installed on the 7900 CPU card. This memory is used to store Function Key definitions, information for buffer sizes, and other important system pointers. The CMOS memory is optional, and comes with a battery-backed supply so that user-defined parameters will be maintained while the system is turned off. This concept is described in detail in the 7900 User's Manual description of the "Thaw" command. If your system does not contain the CMOS option, you will have static RAM installed at these addresses, but the data in this RAM will still correspond to the following table.

This section describes the allocation of CMOS memory in the current version of firmware, TERMEM 1.1. Allocation may change slightly or greatly in future releases. All CMOS is reserved for system use, and any user programs which occupy CMOS do so at the risk of interfering with future system programs.

The CMOS entries which determine buffer sizes should not be altered except through the Thaw command. If these entries do not agree with actual RAM allocation at all times, the system can crash.

Addresses $E40000 through $E40100 are also used by hardware registers in the 7900 system. Accessing these addresses affects CMOS and the hardware as well.

Where appropriate in the following tables, each entry is marked with ".B", ".W", or ".L", to indicate the data size of the entry.

Address		Use
$E40000	.W	Bitmap roll counter
$E40002	.W	X pan register
$E40004	.W	Y pan register
$E40006	.B	X zoom register
$E40007	.B	Y zoom register
$E40008-$E40009		(Reserved)
$E4000A-$E4000F		Raster processor registers
$E40010	.W	Blink select register
$E40012	.W	Plane select register
$E40014	.W	Plane video switch register
$E40016	.W	Color status foreground register
$E40018	.W	Color status background register
$E4001A	.W	Overlay roll counter
$E4001C-$E4001F		(Reserved)
$E40020-$E4003E		Raster processor registers

$E40040-$E4010B		(Reserved)
$E4010C	.L	CMOS verifier long word
$E40110-$E40113		(Reserved)
$E40114	.W	Size of DOS Transient Program Area
$E40116	.W	Size of DOS Buffer
$E40118	.B	Number of active windows
$E40119		(Reserved)
$E4011A	.W	Size of keyboard buffer
$E4011C	.W	Size of Function Key stack (nesting)
$E4011E	.W	Size of RS-232 input buffer
$E40120	.W	Size of RS-232 output buffer
$E40122	.W	Size of RS-449 input buffer
$E40124	.W	Size of RS-449 output buffer
$E40126	.W	Size of Escape code argument stack
$E40128	.W	Size of system stack
$E4012A	.L	Highest RAM address used by system
$E4012E	.L	Pointer to INLINE recall buffer
$E40132	.L	Recall buffer size
$E40136-$E40141		Pointers for INLINE
$E40142	.L	Pointer to start of Function Key buffer
$E40146	.L	Pointer to end of Function Key buffer
$E4014A	.L	Pointer to Case Table
$E4014E-$E4015D		(Reserved)
$E4015E	.L	Address of default program (executed by Boot)
$E40162	.L	Address to search for RAM modules
$E40166-$E40169		TERMEM status flags
$E4016A	.L	Address of Bitmap plot cursor descriptor
$E4016E	.L	Address of Bitmap alpha cursor descriptor
$E40172-$E40179		(Reserved)
$E4017A-$E4017B		INLINE Recall flags
$E4017C-$E401C7		Default Boot string
$E401C8-$E40213		Default Reset string
$E40214-$E4021D		Host EOL sequence
$E4021E	.L	Address of vector-drawn character font
$E40222	.B	RS-232 mode command
$E40223-$E40224		(Reserved)
$E40225	.B	RS-232 handshake flags
$E40226-$E40229		(Reserved)
$E4022A	.B	RS-449 mode command
$E4022B-$E4022C		(Reserved)
$E4022D	.B	RS-449 handshake flags
$E4022E-$E40231		(Reserved)
$E40232-$E407FF		(Reserved)
$E40800-$E408FF		Case Table
$E40900-$E40BFF		Function Key buffer
$E40C00-$E40CFF		INLINE Recall buffer
$E40D00-$E40FFF		(Reserved)

LOW RAM ALLOCATION

The area of RAM between addresses $400 and $FFF is used by the 7900 system for pointers and miscellaneous constants. The area between $1000 and $1C3B is used for DOS tables and pointers. As mentioned earlier, areas marked "Reserved" should be left alone, or risk incompatibility with future releases of software.

Address	Use
$400-$463	Monitor input line
$464-$495	Monitor flags and breakpoint storage
$496-$4E5	Monitor pseudo-register storage
$4E6-$509	Monitor register display formats
$50A-$69B	Monitor stack
$69C-$BFF	(Reserved)
$C00 .L	Pointer to base of TERMEM dispatch tables
$C04 .L	Pointer to Keyboard buffer start
$C08 .L	Keyboard input pointer
$C0C .W	Keyboard buffer character count
$C0E .L	Pointer to Keyboard buffer end
$C12 .W	Joystick X center offset
$C14 .W	Joystick Y center offset
$C16 .W	Joystick Z center offset
$C18-$C1B	Light pen argument list
$C1C .L	Function Key stack pointer
$C20 .L	Pointer to Function Key buffer
$C24 .L	Pointer to Function Key stack
$C28 .L	Pointer to RS-232 input buffer
$C2C .L	Pointer to RS-232 output buffer
$C30 .L	Pointer to RS-449 input buffer
$C34 .L	Pointer to RS-449 output buffer
$C38 .L	Pointer to Esc argument stack
$C3C .L	Pointer to DOS buffer
$C40 .L	Pointer to window table base
$C44 .L	Pointer to top (start) of stack
$C48 .L	Pointer to bottom (end) of stack
$C4C .L	Pointer to start of Create Buffer
$C50 .L	Pointer to end of Create Buffer space
$C54 .L	Pointer to DOS transient area
$C58-$C5D	TERMEM storage for keyboard light data
$C5E-$C61	TERMEM storage for HVS calculations
$C62 .W	Active image planes in system
$C64 .W	Copy of interrupt mask register
$C66 .W	Copy of baud rate generator data
$C68-$C7F	Escape code processor storage area
$C80 .L	Pointer to current character in Create Buffer
$C84 .L	Pointer to end of Create Buffer data (EOF+1)
$C88-$C8D	Joystick data storage area
$C8E .L	Warmstart vector (for USER W)

$C92-$C99	RS232 input ring buffer pointers
$C9A-$CA1	RS232 output ring buffer pointers
$CA2-$CA9	RS449 input ring buffer pointers
$CAA-$CB1	RS449 output ring buffer pointers
$CB2-$FFF	(Reserved)

DOS memory allocation begins here....

$1000-$1009	DOS command block for disk controller
$100A-$103F	Default DOS UFT (details below)
$1040-$113F	Directory buffer space
$1140-$1143	Command block variable space
$1144-$118B	DOS input line buffer
$118C-$11B3	DOS variables and pointers
$11B4 .L	GOADDR: Start address for executable files
$11B8-$11C3	DOS pointers
$11C4 .B	REVN: Revision number of file (not used)
$11C5 .B	DRIVEN: Drive from which last transient came
$11C6 .W	USERN: Password of current user (** = public)
$11C8-$11CD	DOS variables
$11CE .W	FLSTAT: File status used by CLOSE
$11D0 .W	SLASH0: Slash/0 mode flag
$11D2-$11E1	Disk controller variables
$11E2-$1217	TUFT1: Transient UFT #1
$1218-$1243	DOS variables
$1244-$1279	TUFT2: Transient UFT #2
$127A-$1389	DOS variables
$138A-$13FF	Reserved for DOS expansion

THE USER FILE TABLE

DOS maintains a User File Table (UFT) in memory for each file currently being accessed. The UFT contains the file name, password, directory location, size, and various pointers which uniquely identify that file. Whenever DOS reads from or writes to a file, the UFT is updated to show the current file status. The UFT must remain intact as long as a file is in use, or until a new file is closed.

Each UFT occupies 54 bytes. DOS has three UFTs allocated in low RAM, and your program may use these areas to built UFTs, or you may use other RAM.

Each item in the UFT is defined below. If A0 points to the start of a UFT, the item of interest may be accessed with the "Indirect with Displacement" addressing mode, as OFFSET(A0).

Offset in UFT	Description
$00 .W	SLOT: Location of a file in the disk directory.
$02 .L	BUFFP: Memory address to/from which the data transfer will take place on the next read or write.
$06 .L	MBYTES: Number of bytes to transfer on next read or write.
$0A .W	CONTRL: Defines the next operation to be performed. In this word,

Bit	Meaning
9	1 = End of file
8	1=Read, 0=Write
7	1 = Enable retry
6	1 = Enable ECC

$0C .B	DRIVE: Logical unit number of disk drive: 0 and 1 are floppy disks, 2 is the hard disk.
$0D .B	ERROR: Error code of last operation.

$0E .L	BPNTER:	Pointer to current byte in file.
$12 .L	BLNGTH:	Current length of file (goes to zero as file is read in).
$16-$1D	PNAME:	8-character primary filename.
$1E-$20	SNAME:	3-character secondary filename.
$21	REV:	File revision number (not used in DOS 1.4).
$22-$23	PSWRD:	2-character password. Public files are given password "***".
$24 .L	START:	Starting address of file on disk (bytes).
$28 .L	LENGTH:	Length of file on disk (bytes).
$2C .L	ORIGIN:	File origin time/date.
$30 .L	ACCESS:	Time/date of last access.
$34 .W	STATUS:	File attributes as follows...

Bit	Meaning if SET
15	Blind file
8	Active slot
7	Write-protected
6	Delete-protected
4	Free (deleted)
3	Execute-only
1	Odd length
0	Killed

WRITING TRANSIENTS

This section describes the procedure for writing a transient program executable under DOS.

A transient must be located in an appropriate area of RAM, usually the DOS Transient Program Area. This begins at address $1C3C (hex). The length of this area is variable depending on your requirements, but will normally be at least 16K bytes.

```
ORG.L   $1C3C
```

Any RAM used by a transient should be in this same area of memory, or in the DOS Buffer which immediately follows it. RAM allocations are defined by pointers in system RAM and in CMOS, described earlier. Your transient should be careful not to exceed these allocations or the system may become confused.

When a transient is executed, DOS loads it at whatever address it was assembled (DOS does not check whether this is a legal address!). DOS then begins execution at the start address you defined in your END statement. On entry to a transient, A1 points to the first unused character in the command line.

If your transient expects to pick up an argument from the command line, such as a file name, A1 will be pointing to that argument. Note: DOS will only advance A1 past a SINGLE delimiter. If the user typed two spaces, commas, etc., A1 could be pointing to a delimiter.

If the last delimiter found by your program was a colon, assume that another command follows on the command line, and flag this before returning to DOS. This is done by setting D1.B non-zero, and pointing A1 to the first character of the command following the colon. Note that most DOS argument parsing routines automatically bump A1 to the next character after gathering arguments.

Your transient now has control of the system. Using routines provided in the jump tables, your transient can pick up arguments from the command line, open and close files, and control the entire 7900 system. (See Appendix D in the 7900 User's Manual for information on Traps.)

Before returning to DOS, your transient must set D0.B zero if no errors occurred. If you want DOS to report an error, put the error number in D0.B and DOS will display the error message for you. If an error is reported, DOS will not attempt to process any further commands which were entered on the same command line.

If no errors were reported, and you return to DOS with D1.B non-zero, DOS assumes another command is waiting on the input line to be processed. A1 should be pointing to the first character of the next command. Your transient will have to determine whether a colon existed on the command line, possibly by backing up A1 to look for it. (Some DOS routines, such as GETNAM, assist in this process by returning the delimiter character to you).

Summary of registers used by DOS transients:

On entry to a transient:

A1.L points to the first unused character in the command line.

On exit from a transient:

D0.B must equal zero if no error occurred. If D0.B equals zero and D1.B is non-zero, then DOS will attempt to process another command from the command line. A1.L must point to the proper character in the command line.

D0.B must not equal zero if an error occurred. If D0.B is non-zero, DOS assumes it is an error number and will print the corresponding error message. In this case, D1 and A1 are not used, but A0.L must point to the UFT of the file which caused the error (so DOS can report the offending drive number).

A0.L is required if D0.B is non-zero (see above).

D1.B is required if D0.B is zero and another command follows on the command line (see above).

A1.L is required if D0.B is zero and another command follows on the command line (see above).

```
**********************************************************************
*
*
*   Sample transient: READ
*   Command format:  READ <filename>
*
*
*   This transient demonstrates:
*
*       Parsing a file name as a command line argument
*       Opening a file
*       Reading data from the file
*       Exiting from a transient, returning properly to DOS
*
*
*
**********************************************************************
*
*
*   DOS Equates
*
GETNAM  EQU     $80C064         Get file name from input line
RWBYTE  EQU     $80C024         Read/Write bytes to disk
OPEN    EQU     $80C010         Open existing disk file
PRTMSG  EQU     $80C084         Print ASCII string
*
*
*   DOS UFT Equates
*
TUFT1   EQU     $11E2           UFT #1 for transients
U       EQU     0               Base of UFT
SLOT    EQU     U               Logical file number
BUFFP   EQU     SLOT+2          Memory pointer for transfer
MBYTES  EQU     BUFFP+4         # of bytes to transfer
CONTRL  EQU     MBYTES+4        various flags
DRIVE   EQU     CONTRL+2        Drive # (LUN)
ERROR   EQU     DRIVE+1         Error # of last operation
BPNTR   EQU     ERROR+1         Current pointer to file on disk
BLNGTH  EQU     BPNTR+4         Current file length
PNAME   EQU     BLNGTH+4        Primary file name
SNAME   EQU     PNAME+8         Secondary file name
REV     EQU     SNAME+3         Revision level (not used)
PSWRD   EQU     REV+1           File password
START   EQU     PSWRD+2         Points to file start on disk
LENGTH  EQU     START+4         Length of file on disk
ORIGIN  EQU     LENGTH+4        Time/Date of file origin
ACCESS  EQU     ORIGIN+4        Time/Date of last access
STATUS  EQU     ACCESS+4        File status
*
```

```
*
*   TERMEM Equates
*
CHAROUT EQU     $800008         Character-out
CHARIN  EQU     $80000C         Character-in
CTRLIN  EQU     $800014         Character-in with Esc processing
CTRLOUT EQU     $800010         Character-out with Esc proc
DOSBUFP EQU     $C3C            Pointer to DOS buffer
DOSBUFZ EQU     $E40116         Size of DOS buffer
*
*
*   ASCII Equates
*
CR      EQU     13              Carriage return
LF      EQU     10              Line feed
DEL     EQU     $7F             Delete
XOFF    EQU     $13             X-off (Control-S)
*
*
*****************************************************************
*
*   The transient begins execution here.
*

        ORG.L   $1C3C           We run in DOS transient area
        LLEN    132             Printer is 132 columns wide

READ    EQU     IPC
        MOVE.L  #TUFT1,A0       Point to UFT to use
        MOVE.L  #DefName,A2     Point to default name: '*.SRC'
        JSR     GETNAM          Get file name into UFT.
        MOVE.L  A1,SaveA1       Save cmd line pointer for DOS
        TST.B   D0
        BNE     READerr         GETNAM found an error.

        CMPI.B  #':',D1         Was command delimiter a colon?
        SEQ     Another         If YES, another command follows.
*
*
*   The UFT now contains a filename, parsed by GETNAM.  (If
*   no filename was entered, the default filename remains
*   in the UFT, *.SRC)
*
*
        JSR     OPEN            Attempt to open the file.
        TST.B   D0
        BNE     READerr         OPEN found an error.
*
```

```
*
*   OPEN has now provided the UFT with details of the file
*   such as its length and disk location.
*
*
READ1   MOVE.L  BLNGTH-U(A0),D4     D4.L = file length remaining
        BEQ.S   READex              If zero, file is empty now.

        MOVE.L  DOSBUFP,BUFFP-U(A0)     Put data in DOS buffer
        CLR.L   D5
        MOVE.W  DOSBUFZ,D5              D5.L = size of DOS buffer
*
*   Check if bytes left in file will fit into DOS buffer.
*
        CMP.L   D4,D5           Will it all fit?
        BHI.S   READ2           Yes.
        MOVE.L  D5,D4           NO, so only read what WILL fit.

READ2   MOVE.L  D4,MBYTES-U(A0) Read this many bytes
        BSET    #0,CONTRL-U(A0) Say READ
        JSR     RWBYTE          Go and read bytes from disk.
        TST.B   D0
        BNE     READerr         Error from RWBYTE.

*
*   We have read bytes into the DOS buffer.  D4 is the number
*   of bytes.
*
        MOVE.L  DOSBUFP,A2      Point to the data
        SUBQ.L  #1,D4           Decrement count for DBRA below
        CLR.W   D1              Specify Device 0 for CHAROUT
*
*   Display the bytes.
*
READ3   MOVE.B  (A2)+,D0        Get a byte of data
        JSR     CHAROUT         Display it
        CMP.B   #CR,D0          Was it a Carriage Return?
        BNE.S   READ4           No
        MOVEQ.L #LF,D0
        JSR     CHAROUT         Follow Return with Line Feed.
*
*   After printing each character, check if we should pause.
*
READ4   JSR     CHARIN          Check the keyboard.
        BEQ.S   READ5           No key was hit.
        CMP.B   #DEL,D0         Did he hit DELETE?
        BEQ     READex          Yes.... quit.
        CMP.B   #XOFF,D0        Did he hit Ctrl-S?
        BNE.S   READ5           No, so ignore the key.
*
```

```
*
*   Control-S was hit, so pause.
*
READp   JSR     CHARIN          Wait for another key
        BEQ.S   READp
        CMP.B   #DEL,D0         DELETE?
        BEQ     READex          Yes, so quit.
READ5   DBRA    D4,READ3        Go display more data.
        BRA     READ1           Go get more data from disk.


*****************************************************************
*
*   Come here for normal exit from the transient.
*
READex  CLR.B   D0              Flag NO error and fall into exit.

*
*   Come here to go back to DOS with error message.
*   D0.B holds the error code.
*
READerr CLR.W   D1              Logical Device 0 for CHAROUT
        MOVE.W  D0,-(SP)        Save the error flag
        MOVE.B  #LF,D0          Print a line feed
        JSR     CHAROUT         Or two...
        JSR     CHAROUT
        MOVE.B  Another,D1      Flag if another command existed
        MOVE.L  SaveA1,A1       Restore command line pointer
        MOVE.W  (SP)+,D0        Restore the error flag
        RTS                     Back to you, DOS!


DefName DC.B    '*      SRC'    Default name for UFT (11 chars)
        DS.L    0               Align even addresses after DC.B

SaveA1  DS.L    4               Place to save A1
Another DS.B    1               Flag for colon on command line
        DS.L    0               Align even addresses after DS.B
*
*
*
*
        END     READ            Begin execution at label READ
*
*
*
```

CUSTOM CHARACTER SETS

The 7900 allows user-defined character sets to be used in the Bitmap in place of the two character sets supplied with the system. An entry in each Window Table (Charadr) points to the base of the character set for that window, and the size of the font (X by Y pixels) may also be defined for each window individually. The character font dimensions may be up to 16 in the X direction, and 256 in the Y direction.

Since the character set address for a window is stored in the Window Table, it will default back to the normal character set whenever Boot or Soft Boot is executed.

The character set for the Overlay is stored in high-speed PROM, and is not alterable through software.

The following program is a module, designed to be linked into the 7900 system software. It will install a custom character set in any window which receives a "Mode i" command. This program is an example ONLY. It does not include a complete character set definition. The data set which should accompany this program would be too long to fit into the standard 7900 memory, unless you do one of the following: (1) change the ORG address, which requires changing address MDLE with the Thaw command, (2) change the height of the character set to reduce the data required, or (3) install additional memory above address $20000.

```
*
*    Sample module to install a new character set.
*    The set is installed in a window by the command:
*
*        MODE i
*
*    To return to the standard set, use  SOFT BOOT.
*
*

*
*    CGC 7900 equates...
*

Mode    EQU    1          MODE is Control-A

CharXZ  EQU    $78        Offset in window table for X raster size
CharYZ  EQU    $7A        Y raster size
CharDX  EQU    $7C        X intercharcter spacing (step)
CharDY  EQU    $7E        Y intercharcter spacing
Charadr EQU    $9C        Address of font for this window


*
*
*
        ORG.L   $1F000          This is the address called "MDLE"
*                               as specified by the Thaw command.
*
        DC.L    'MDLE'          This header must be present if
*                               the module is located in RAM.
*
        DC.W    CharEnd-IPC     This is the length of the
*                               module (including the
*                               character set at end)
*
        DC.B    Mode,'i'        We are executed when the user
*                               types this sequence.
*
        DC.W    $0000           We require no arguments.
        DC.W    $0000

*
*    Execution begins here after MODE i is entered.
*    TERMEM preloads A0 with the base of the window table.
*

        MOVE.W    #6,CharXZ(A0)     Load character size (X)
        MOVE.W    #8,CharYZ(A0)                         (Y)
        MOVE.W    #6,CharXZ(A0)     Load step size (X)
        MOVE.W    #8,CharYZ(A0)                    (Y)
        MOVE.L    #BASE,Charadr(A0)     Load address of font
        RTS                             That's all!
```

```
*****************************************************************
*
*    What follows is the character font definition.
*
*    It is arranged as 128 regular characters, followed by 128
*    A7 characters.  All 256 characters should be defined.
*
*    Assume the font is set in a field X by Y.  (Default is
*    X = 6 and Y = 8.)  Then each character requires 8 words
*    of data, one for each Y scan; and 6 bits of each word
*    are used.  The active 6 bits are left-justified in the
*    16-bit word.
*
*
*    Example: the character "A"
*
*                                   bit #
*
*                                   111111              Hex value
*                                   5432109876543210
*
*                              1    .XXX............     $7000
*                              2    X...X...........     $8800
*                              3    X...X...........     $8800
*                 Word #       4    XXXXX...........     $F800
*           (one per Y scan)   5    X...X...........     $8800
*                              6    X...X...........     $8800
*                              7    X...X...........     $8800
*                              8    ................     $0000
*
*
*    The X by Y field must include any necessary
*    spacing between characters or between lines.
*
*
*    Memory requirements:
*
*    This module will require 256 * Y words, plus 23 words of
*    overhead for the preceding code.  This much memory must
*    exist between address "MDLE" and the physical end of memory.
*    If necessary, use Thaw to alter the address of "MDLE" (be
*    sure to put it where nothing will stomp on it!)
*
*
```

```
*
*
*    This Scale Factor will left-justify a 6-bit number in a
*    16-bit field, by shifting 10 bits.  All of the numbers in
*    this database are 6-bit for convenience, and S justifies
*    them properly.
*
S       EQU    1024
*
*
BASE    EQU    IPC          This is where it all begins.
*
*    This is the regular character set.
*
    DC.W    $24*S,$34*S,$3C*S,$2C*S,$24*S,$04*S,$04*S,$07*S     ^@
    DC.W    $0C*S,$10*S,$08*S,$05*S,$1D*S,$07*S,$05*S,$05*S     ^A
*     .
*     .    etc. for regular set.
*     .
    DC.W    $01*S,$0E*S,$10*S,$00*S,$00*S,$00*S,$00*S,$00*S      ~
    DC.W    $0A*S,$15*S,$0A*S,$15*S,$0A*S,$15*S,$0A*S,$00*S     DL
*
*
*
*
*    Alternate (A7) character set begins here.
*
    DC.W    $1F*S,$15*S,$15*S,$1F*S,$15*S,$15*S,$1F*S,$00*S     ^@
    DC.W    $08*S,$15*S,$02*S,$08*S,$15*S,$02*S,$00*S,$00*S     ^A
*     .
*     .    etc. for A7 set.
*     .
    DC.W    $01*S,$02*S,$02*S,$04*S,$08*S,$10*S,$10*S,$20*S      ~
    DC.W    $00*S,$00*S,$00*S,$00*S,$00*S,$00*S,$00*S,$3F*S     DL
*
*

CharEnd EQU     IPC      This is the end of our module.
        DC.L    -1,-1    No more modules follow this one.

        END
*
*
*
*
*
```

INSTALLING A NEW CURSOR

The 7900 Bitmap cursors, Plot and Alpha, are each described by a set of data. This set is pointed to by pointers in CMOS, one pointer for the Plot cursor and one for the Alpha cursor.

```
Plotcur   EQU   $E4016A
Alphcur   EQU   $E4016E
```

The cursor descriptor data is a list of up to 32 long words. Each long word describes the displacement of one pixel of the cursor, with respect to the center pixel of the cursor. The list is terminated with a zero word. Since this zero word is part of the descriptor, the center pixel of the cursor is always ON.

The displacements are given as addresses in Bitmap memory. Each pixel in Bitmap memory corresponds to a word (two bytes) of memory, so an X displacement of one pixel is produced by an address displacement of two. (Positive X displacement is to the right.) Similarly, a Y displacement of one pixel corresponds to an address change of 2048 bytes (1024 pixels per Y line of the screen, times two bytes per pixel). A positive Y displacement is in the down direction.

A sample cursor might look like this, where X's correspond to pixels included in the cursor:

```
 X
XXX
 X
```

The data list for this cursor would be:

```
   +2  (the pixel to the right of center)
   -2  (the pixel to the left of center)
+2048  (the pixel below center)
-2048  (the pixel above center)
    0  (the center pixel, and end of the list)
```

To install a new cursor, first define it in the form above. Store this data in memory. Then, alter the pointer in CMOS (either Plotcur or Alphcur) so that it points to your data. Note that if you store your cursor in RAM other than CMOS, the description will vanish when system power is turned off, but the CMOS pointer will remain! This will cause you to have NO cursor at all. To reload CMOS defaults, use CTRL SHIFT M1 M2 RESET.

```
*****************************************************************
*
*     Sample program PUTCURS
*
*     Installs a new cursor as the Bitmap plot cursor.
*
*     This program stores its cursor descriptor in upper CMOS
*     memory, unused by current 7900 sofware.  This may not be
*     compatible with future 7900 releases.
*
*
*
         ORG.L   $1C3C          We run in DOS area

PUTCURS  MOVE.L  #HiCMOS,A2     Point to some unused CMOS
         MOVE.L  #Cursor,A3     Point to our new cursor descriptor

PUTloop  MOVE.L  (A3)+,(A2)+    Copy a long word into CMOS
         TST.L   -4(A2)         Was it zero?
         BNE.S   PUTloop        No, continue copying

         MOVE.L  #HiCMOS,Plotcur   Set up pointer to new cursor

         CLR.L   D0             Flag no error occurred
         CLR.L   D1             (We don't check for colon on line)
         RTS                    Return to DOS


HiCMOS   EQU     $E40E00        CMOS area (unused in TERMEM 1.1)
Plotcur  EQU     $E4016A        Plot cursor pointer

Cursor   DC.L    -4*1024        New cursor descriptor
         DC.L    -4*1024+2
         DC.L    -4*1024-2
         DC.L    -2*1024-4
         DC.L    -2*1024+4
         DC.L    -4
         DC.L    +4
         DC.L    +6
         DC.L    +8
         DC.L    -2*1024+8
         DC.L    -2*1024+10
         DC.L    -2*1024+12
         DC.L    -2*1024+14
         DC.L    -4*1024+10
         DC.L    -4*1024+12
         DC.L    -4*1024+14
         DC.L    -4*1024+16
         DC.L    4*1024
```

```
        DC.L    4*1024+2
        DC.L    4*1024-2
        DC.L    2*1024-4
        DC.L    2*1024+4
        DC.L    2*1024+8
        DC.L    2*1024+10
        DC.L    2*1024+12
        DC.L    2*1024+14
        DC.L    4*1024+10
        DC.L    4*1024+12
        DC.L    4*1024+14
        DC.L    4*1024+16
        DC.L    0               (end of list)

        END     PUTCURS
*
*
*
```

DOS ERROR MESSAGES

The following errors may be reported by DOS. To force DOS to print an error message, load the error number into D0.B before returning to DOS.

Error (hex)	Message
01	No index signal detected
02	No seek complete
03	Write fault
04	Drive not ready
05	Drive not selected
06	No track 000 detected
10	ID read error
11	Uncorrectable data error found during a read
12	ID address mark not found
13	Data address mark not found
14	Block not found
15	Seek error
16	No host acknowledgement
17	Diskette write protected
18	Data field error found and corrected
19	Bad track found
1A	Format error
20	Invalid disk controller command
21	Illegal logical block address
22	Illegal function for the specified drive
30	Diagnostic RAM error
40	Disk controller not ready
41	Controller time out error
42	Unable to determine controller error
43	Undefined controller state
44	Controller protocol sequence error
50	Undefined load error state
51	Record count error
52	Checksum error
53	Premature EOF during load
54	DOS buffer too small
55	Transient program size too small
60	End of file reached
61	File is write protected
62	Attempted to read thru density barrier
63	Attempted to transfer data on odd address

70 Unable to find requested file

80 Unable to create new file space

90 Unable to close requested file

A0 Empty slot found
A1 Unable to update the directory

B1 No run address
B2 Unable to find disk name
B3 Argument error
B4 Attempt to access a non-existant drive
B5 Unable to initialize drive 1
B6 Unable to initialize drive 2
B7 Syntax error! Missing argument

C0 Premature format termination
C1 Error mapping routine not implemented
C2 Unable to fetch this file
C3 File is delete protected
C4 File type error
C5 File is execute only
C6 File is to big to append
C7 Insufficient stack size
C8 /0 mode is not allowed in argument filenames