2 8 0 1 0 0 DMXMON

DATAMOVER CROSS MONITOR

REFERENCE MANUAL

SEPTEMBER, 1982

TABLE OF CONTENTS

1. DMXMON INTRODUCTION

The name DMXMON is an acronym derived from the words: DataMover Cross MONitor. A cross monitor is a program that facilitates the preparation, execution, and debugging of programs on a slave processor much like the resident operating system facilitates these activities on the main system processor. When under the control of DMXMON, the MTU-130 actually appears to be a 68000-based system. Conversely, a running 68000 program is "fooled" into believing that it has direct access to all MTU-130 peripherals by channeling these accesses through DMXMON. Most importantly, a 68000 program is given access to the powerful file and I/O management functions of CODOS (which runs on the 6502) just as easily as if CODOS was running on the 68000. Thus with the DMXMON cross monitor, the user/programmer can interact with the 68000 slave processor just as effectively as if it were in fact the main processor and often even more effectively. In fact, when a 68000 program is running, the 6502 serves as a peripheral processor, freeing the 68000 from I/O chores and improving throughput.

In the discussion that follows, a working knowledge of CODOS concepts and commands as well as fundamental machine language concepts is assumed. Additionally, some terminology peculiar to the 68000 microprocessor will be used. It is thus recommended that the CODOS, Datamover, and MC68000 microprocessor manuals be handy while studying DMXMON. It is also helpful to actually try the examples using an MTU-130 equipped with a Datamover.

1.1 SUMMARY OF DMXMON CAPABILITIES

DMXMON was designed to make all of the MTU-130 facilities that are normally available to the 6502 programmer also available to the 68000 programmer. There are some additional facilities made possible by the Datamover's shared memory slave processor architecture as well. DMXMON is truly an operating system itself so there is no need to obtain and learn a new operating system just to tap the power of the 68000 microprocessor.

For the user, DMXMON offers the complete array of CODOS file maintenance commands such as FILES, RENAME, DISK, KILL, TYPE, and others. These operate exactly like their CODOS counterparts and in fact are implemented by "passing the command through" to CODOS for execution. The user can also execute the full array of standard utilities, the only requirement being that they do not interfere with memory used by DMXMON (see the appendix for memory maps). A user written 68000 program is executed simply by typing its name along with any required arguments, just as with a user 6502 program. Thus file maintenance and running 68000 programs is just as simple and straightforward with DMXMON as 6502 programs are with CODOS. Most importantly, the commands and overall operational philosophy is unchanged, meaning that the user's knowledge about CODOS is directly applicable to the Datamover and DMXMON.

For the programmer, DMXMON offers an extensive array of over 90 "supervisor calls" (SVC) that allow access to all of the CODOS and MTU-130 I/O functions. A 68000 TRAP instruction followed by an ID number and arguments in the registers is the supervisor call mechanism. Since the ID numbers will never be changed, 68000 programs can be independent of any changes that may be later made to DMXMON.

For simplicity in programming, the standard start-wait-complete form of the SVCs may be used. With this form, the SVC processor maintains control until the requested operation is completed. For greater throughput, the "start only" form may be used which will return control to the user's 68000 program as soon as the operation is started. Computation may then overlap execution of the I/O operation. Later, when the SVC's results are required, a "wait-complete" SVC may be executed.

Although DMXMON is basically a single-user single-task type of operating system like CODOS, several features useful for multitasking-like operation are provided. For example, a timer may be started which will interrupt the 68000 program at a set periodic rate or automatically increment a long word in 68000 memory. When using start-only SVCs, an interrupt can be given when the SVC is complete. Even an interrupting keyboard may be enabled whereby every keystroke entered will interrupt the 68000. When enabled, the interrupting keyboard remains functional even during other I/O operations such as disk access. An interrupt handler in DMXMON sorts out these various interrupts and automatically vectors to the user's service routines. Note that these extended features will never "get in the way" of minimum complexity programming if they are not used. Contrast this with most other 68000 based systems that seem to require a degree in Computer Science just to do simple I/O programming.

One of the most significant features of DMXMON is its ability to rapidly copy a portion of Datamover memory into the MTU-130's display memory. Total flexibility is offered whereby a rectangular "window" of any size in 68000 simulated display memory may be copied into any size and position "viewport" in real 6502 display memory. This can occur either on demand by the 68000 program or may be performed automatically by the 6502 in the background with virtually no slowdown of the 68000 program. With the Datamover's extremely large memory capacity, it is quite practical to define a virtual display space of 1000x1000 pixels or more (that would be 128K or 1/2 the Datamover's standard memory) and display a selected portion up to 480x256 pixels in less than 1/4 second.

For the debugger, DMXMON offers an extensive array of memory manipulation commands, execution mode error messages, and program breakpoints. All of the standard CODOS memory manipulation commands such as SET, FILL, HUNT, DUMP, COMPARE, etc. are provided, the only difference being that they operate on 68000 memory. Full 24 bit addresses are recognized as arguments of these commands and 32 bit arithmetic is utilized when evaluating expressions. In addition, the contents of all 68000 registers may be displayed and changed with the REG command. The 68000 microprocessor recognizes a number of "trap" conditions during execution such as illegal instruction, divide by zero, addressing error, and others. These are reported by DMXMON with an English error message showing the problem location and register contents. Up to 4 breakpoints may be set in a 68000 program which will stop execution and print the register contents when encountered. Execution may be resumed after the breakpoint with all registers and status restored. The 68000 program trace feature is not used by DMXMON and therefore is available to user programs. User programs may run in either the 68000's supervisor mode or user mode with the default being supervisor mode.

1.2 CONSOLE MODE

Like any system monitor, DMXMON operates in two different modes. In the console mode, the DMXMON> prompt is displayed on the console which signifies that it is waiting for a command from the operator. This is exactly analogous to the CODOS> prompt seen in normal MTU-130 operation. Below is a summary list of the available DMXMON commands. They are described in detail in later sections. Note that all commands in this list that deal with memory deal with Datamover (68000) memory, not 6502 memory.

COMMAND	FUNCTION
BP [<address>]	Set or clear a 68000 program breakpoint
BP?	Display currently set breakpoints
BYE	Return control to CODOS
CALC <expression>	Display value of <expression> in hex and decimal
CODOS	Return control to CODOS
COMPARE <arg like CODOS>	Compare two memory blocks for equality
COPY <args like CODOS>	Copy memory contents from one area to another
DUMP <args like CODOS>	Display memory contents in hex and ASCII
FILL <args like CODOS>	Fill memory block with a constant
GET <args like CODOS>	Load file into memory and set PC to entrypoint
GETLOC <args like CODOS>	Display load addresses and entrypoint of file
GO <args like CODOS>	Start 68000 program execution
GROUP <group #>	Set group select register to specified value
HUNT <args like CODOS>	Search memory block for specified values
NEXT <args like CODOS>	Resume 68000 program execution after breakpoint
REG [<reg id> <cnts>]	Display or alter contents of 68000 registers
RESET	Reset the 68000 then restore 68000 DMXMON
SAVE <args like CODOS>	Save memory contents as a 68000 loadable file
SET <args like CODOS>	Store specified values into memory
? <expression>	Display value of <expression>, same as CALC

In addition to the Datamover specific DMXMON commands listed above, a number of frequently used CODOS commands are recognized as such and are passed through to CODOS for execution. These "pass-through" commands are listed below:

COMMAND	FUNCTION
ASSIGN	Assign a channel to a file or device
BEGINOF	Position a channel to beginning of data
BOOT	Exit DMXMON and reload CODOS operating system
CLOSE	Flush buffers and allow removal of a diskette
COPYF	Copy specified files
DATE	Set the date
DELETE	Delete specified files immediately
DIR	Display full name and size of specified files
DISK	Display remaining space and VSN of open drives
DO	Read commands from a file

DRIVE	Set default disk drive
ENDOF	Position a channel to end of data
FILES	Display all filenames on a specified drive
FORMAT	Erase and prepare a new disk
FREE	Unassign a channel and close its associated file
KILL	Delete specified files with operator verification
LOCK	Write protect specified file
MSG	Display a message on the console
ONKEY	Define a function key legend and substitution string
OPEN	Allow access to disk in specified drive
PROTECT	Disallow alteration of memory used by CODOS
RENAME	Change the name of an existing file
TYPE	Display contents of a text file
UNLOCK	Allow write or delete of specified file
UNPROTECT	Allow alteration of memory used by CODOS
WAIT	Wait for specified time and continue

Other CODOS commands and standard utility commands can also be passed through (even if the name is the same as a DMXMON command) by preceeding the command with a period. Thus the command .DUMP would display the contents of 6502 memory (standard CODOS DUMP) whereas just plain DUMP would display the contents of 68000 memory.

Finally, when 68000 programs are saved as a file by DMXMON using the SAVE command, they become "user defined DMXMON commands" which can be executed simply by typing their names. This is exactly analogous to saving 6502 programs with CODOS which then become user defined CODOS commands.

It is also possible to set up "job files" of DMXMON commands which can be executed with a DO command. The job file may contain any DMXMON command, pass-through command, or user defined command and the effect will be same as if the commands were typed in from the console.

Numerous errors are possible when commands are executed. When DMXMON detects an error, the command is aborted and an error message is printed on the console. When CODOS detects an error, the standard CODOS error message is printed and control is automatically returned to DMXMON which may display an additional message. In either case, another command may be entered when the DMXMON> prompt appears.

All of the standard CODOS line editing functions are available when entering a command. These include cntl/B to recall previous commands, cntl/R to redisplay the current command, and function key string substitution. In addition, all of the standard console display control functions such as cntl/S (stop display), cntl/Q (resume display), and cntl/C (abort command) are also provided.

1.3 EXECUTION MODE

DMXMON is in "execution mode" when a 68000 user program is actually running. When in execution mode, the portion of DMXMON resident in the 6502 is waiting for an operation request from the 68000 portion of DMXMON. Supervisor calls from the user program are interpreted by 68000 DMXMON and then passed to 6502 DMXMON for execution. If enabled, the display will be automatically refreshed from 68000 memory during execution mode.

Transition from console mode to execution mode is triggered when a GO or NEXT command is executed. It is also triggered when an unrecognized command that is the name of a 68000 loadable file is encountered. In any case, the 68000 hardware registers are first loaded from a save area in memory and then a JSR (JMP for NEXT) is taken to the specified address or PC contents or user command entrypoint. DMXMON then enters execution mode.

Transition from execution mode to console mode may be triggered in a number of ways. The normal way is for the 68000 program to finish its task and exit with an RTS instruction or execute one of the two exit supervisor calls. Another way is for the program to encounter a breakpoint set by the operator. Any error condition detected either by the 68000 hardware or by SVC execution will print a message, the register contents, and then enter console mode. The operator may also force an exit at any time by pressing the INT key.

Four classes of supervisor calls are available to the 68000 programmer while in execution mode. The first class consists of 23 CODOS derived SVCs plus two additional "systems related" SVCs. CODOS SVCs related to number conversion and the 16 bit pseudo processor are not included since the 68000's more powerful instruction set makes these functions trivial to program directly. One of the added SVCs will return the time of day as 8 characters (requires the MultI/O board) while the other returns the highest available Datamover memory address.

The second class includes 30 SVCs related to operating the keyboard and character display directly. For regular "stream I/O" it is usually easier to do the I/O through CODOS channels, but for for special purposes such as editors and menu selection programs, these SVCs allow total control of the keyboard and display. There is even a line editing SVC and an SVC for generating an arbitrary tone with specified pitch, volume, and duration.

The third class consists of 21 SVCs for graphic data input and display. Full access to the MTU-130's light pen, 480x256 pixel mapped display, and key controlled cross-hair graphic cursor is available. SVCs are provided for drawing, erasing, and flipping lines, dashed lines, and points using either absolute or relative coordinates. The light pen SVCs return a hit flag and X and Y coordinates with 1 pixel resolution.

The last class of SVCs includes 18 functions that are unique to DMXMON and the Datamover. These are summarized below and described in detail in section 3.

SVC NAME	FUNCTION
READM	Read an arbitrary 6502 memory location
WRITEM	Write an arbitrary 6502 memory location
BLKRD	Copy a block of 6502 memory to 68000 memory
BLKWRT	Copy a block of 68000 memory to 6502 memory
CALLM	Call an arbitrary 6502 subroutine
SETWD68	Set 68000 display window parameters
SETVP65	Set 6502 display viewport parameters
RFSH1	Refresh the 6502 viewport from the 68000 window once
RFSHSR	Start automatic refresh of the 6502 viewport
RFSHSP	Stop automatic refresh
CPYDSP	Copy 6502 viewport to 68000 window
WAIT	Wait for a specified time
CLKON	Start periodic interrupt at specified rate
CLKOFF	Stop periodic interrupt
CNTON	Start periodic in-memory counter update
CNTOFF	Stop in-memory counter update
KYBON	Start the interrupting keyboard
KYBOFF	Stop the interrupting keyboard

2. CONSOLE MODE COMMANDS

2.1 STARTING AND EXITING DMXMON

Assuming that a disk with DMXMON written on it is in drive 0, it may be started simply by entering:

```
DMXMON
```

in response to a CODOS> prompt. You may optionally follow the DMXMON command verb with any other command and it will be executed immediately after DMXMON is loaded. After DMXMON is loaded (and the following command, if present, is executed), it will display:

```
DATAMOVER RUNNING
DMXMON>
```

and wait for a command. The DATAMOVER RUNNING message indicates that the Datamover was successfully reset and that the 68000 portion of DMXMON is responding correctly. If the Datamover board does not start up properly, a message to that effect is printed instead but DMXMON will still be responsive to commands (except for GO and NEXT of course), presumably for troubleshooting. In most respects, DMXMON commands perform the same function as CODOS commands of the same name. The difference is that they refer to the Datamover's memory instead of regular 6502 memory.

You may exit DMXMON and return to CODOS by entering BYE and a carriage return, or by entering CODOS and a carriage return. When in console mode, the INT key will also exit DMXMON and enter CODOS. All three methods of exit are equivalent. Since virtually all CODOS functions and commands can also be executed from DMXMON, there is little need to leave DMXMON unless your work with the 68000 is complete or you wish to run a 6502 program that uses most of memory such as the Editor, Assembler, or BASIC.

2.2 VALUES AND EXPRESSIONS

Many DMXMON commands require numeric values for arguments. In this case, either decimal or hexadecimal values may be used. Unless otherwise indicated, all numeric arguments are assumed to be in hexadecimal. To specify a decimal argument, use the prefix ".". If desired, the $ prefix can be used to clarify hex values.

An arithmetic expression can be used anywhere a numeric value is called for, except for disk drive numbers. Arithmetic expressions may be formed using the usual operators: "+", "-", "*", "/", and "\". "\" is the remainder operator. All expressions are evaluated left-to-right without any operator precedence. The value entered may not exceed 4294967295 decimal ($FFFFFFFF) or be less that -2147483648 decimal ($80000000) including any intermediate point in the calculation. There is no error message if overflow occurs but the result will be truncated to 32 bits. Note that 32 bit arithmetic applies only to DMXMON unique commands. CODOS pass-through commands continue to use 16 bit arithmetic since they are interpreted by CODOS. The following examples illustrate the evaluation of numeric expressions:

```
100         evaluates as 256 decimal (100 hex)
.100        evaluates as 100 decimal (64 hex)
B+ 10       evaluates as 27 decimal (1B hex). Blanks are ignored after operators
1+.10*3     evaluates as 33 decimal (21 hex)
$1498/.256 evaluates as 20 decimal (14 hex)
40BC\100+1 evaluates as 177 decimal (BD hex)
-.37 evaluates as -37 decimal or 4294967259 decimal (FFFFFFDB hex)
```

Where single byte values are required such as in the SET, FILL, and HUNT commands, an ASCII character may be specified by enclosing it in quotes, either single quotes (') or double quotes ("). The same kind of quotes must be used on each side. When an ASCII character is used, it must stand alone, it cannot be used as a value in an expression.

Memory addresses in DMXMON always refer to the memory onboard the Datamover. These addresses are always byte addresses which is the 68000 convention. In terms of 16 bit words, even addresses refer to the left byte and odd addresses refer to the right byte. The 256K of Datamover memory apppears to be contiguous from $00000 to $3FFFF (up to $FFFFF if the full 768K memory expansion is added). Thus when using DMXMON commands, you view the memory just as the 68000 views it, as one big contiguous chunk. There is no error message if you attempt to access non-existant memory. Addresses greater than $FFFFF (1M byte) are truncated mod $100000.

2.3 DMXMON COMMANDS

Since DMXMON in command mode is primarily a program preparation and debugging tool, most of its commands deal with the convenient manipulation of Datamover memory. You will find most of these commands to be very similar if not identical to their CODOS counterparts.

2.3.1 BP

PURPOSE: To set a program breakpoint for debugging purposes.

SYNTAX: BP [<address>]
BP?

ARGUMENTS: <address> = address of first word of instruction at which breakpoint is desired.

EXAMPLES: BP 420A

will set a breakpoint at address $00420A. Following a GO or NEXT command, when program execution reaches 420A, control will be returned to DMXMON which clears the breakpoint and then prints the contents of all registers (see the REG command description for the format of the register printout). Up to 4 breakpoints may be set simultaneously.

BP will clear all breakpoints.

BP? will print the addresses of all active breakpoints.

NOTES: 1. BP works by temporarily replacing the op-code at the indicated location with a TRAP 0 instruction. The original op-code is replaced when the breakpoint is cleared.

2. Breakpoints may be placed anywhere, even at Supervisor Calls. They should only be placed at the op-code word of an instruction and must therefore be at even addresses.

3. The register printout is preceeded by the keyword -BREAKPOINT- to indicate that entry into DMXMON was through a breakpoint.

2.3.2 BYE

PURPOSE: To exit DMXMON and return to CODOS.

SYNTAX: BYE

EXAMPLE: BYE

will exit DMXMON and return to CODOS. Before exit, the 6502 interrupt vectors, cntl/C vector, and CODOS error processing is restored to normal. If timer or keyboard interrupts had been started by a 68000 program, they will be stopped. The CODOS command or the INT key (when in console mode) can also be used to exit DMXMON.

2.3.3 CALC

PURPOSE: To evaluate an expression and display it in hex and decimal.

SYNTAX: CALC <expression>
? <expression>

ARGUMENTS: <expression> = any valid expression as described in section 2.2.

EXAMPLES:

CALC .87*1000	will display:	$00057000 = 356352
CALC -.27532	will display:	$FFFF9474 = 4294939764 or -27532
?C1345	will display:	$000C1345 = 791365

NOTES: 1. Since numbers with the most significant bit set may be + or - (depending on usage), both representations are printed.

2.3.4 CODOS

The CODOS command is identical to the BYE command described in section 2.3.2.

2.3.5 COMPARE

PURPOSE: To determine if two blocks of memory are identical.

SYNTAX: COMAPRE <from> <to> <dest>

ARGUMENTS: <from> = Starting address of first block
<to> = Ending address of first block
<dest> = Starting address of second block

EXAMPLE: COMPARE C000 12FFF 15000

will compare every byte of the block of memory from $C000 to $12FFF to the corresponding bytes in the block from $15000 to $1BFFF. If the blocks are identical, DMXMON will display SAME. If the blocks differ, the address and content of the first byte which differs will be displayed and the comparison will terminate.

NOTES: 1. The comparison proceeds from the low address upwards.

2. The two memory blocks may overlap thus allowing one to search a memory block for a byte that differs from the rest.

2.3.6 COPY

PURPOSE: To copy a block of memory to another memory location.

SYNTAX: COPY <from> <to> <dest>

ARGUMENTS: <from> = Starting address of the block to be copied.
<to> = Ending address of the block to be copied.
<dest> = Starting address of destination of the copy.

EXAMPLE: COPY 100 2FF 2000 Copies $100 - $2FF to $2000 - $21FF
COPY 3000 3900 3000+.50 Moves the block from $3000 - $3900 up 50 bytes

NOTES: 1. The block to be copied may be any size.

2. The destination for the copy may overlap the block being copied. The direction of copy is automatically adjusted so that the block is correctly moved.

3. The direction of copy may be reversed by swapping the <from> and <to> arguments.

2.3.7 DUMP

PURPOSE: To display the contents of a block of memory in hex and ASCII.

SYNTAX: DUMP <from> <to> <device>
<channel>

ARGUMENTS: <from> = desired starting address
<to> = desired ending address
<device> = desired device on which to display the output. Defaults to the console.
<channel> = desired channel on which to display the output.

EXAMPLES: DUMP 1000 Displays 16 bytes of Datamover memory starting at $1000.
DUMP 2100 211A Displays memory starting at $2100 and will include memory through $211A.
DUMP 0 FFFF P Dumps 64K bytes starting at 0 The display will be output to the printer.

The display produced by these examples will look similar to:

```
        0  1  2  3  4  5  6  7    8  9  A  B  C  D  E  F
002100  34 77 D7 4B 20 00 56 78   4F 4B 20 46 45 4C 4C 41   4w.H..Vx OK.FELLA
002110  55 67 09 42 59 45 00 03   03 03 20 00 00 00 10 20   Ug.BYE.. ........
```

Of course the actual values displayed will depend on the contents of memory. The 16 rightmost characters on each line are the ASCII characters for the line, with each non-displayable character converted to ".", including blanks.

NOTES: 1. A complete line is always displayed even if the <to> address is in the middle of the line.

2. Cntl/S may be used to temporarily suspend the display and cntl/Q will restart it. Cntl/C will terminate the display and return to DMXMON.

2.3.8 FILL

PURPOSE: To fill a block of memory with a constant.

SYNTAX: FILL <from> <to> [=] "<character>" / <value> / '<character>'

ARGUMENTS: <from> = Desired starting address to be filled.
<to> = Desired ending address to be filled.
<value> = Numeric constant to be deposited into each byte.
<character> = Single ASCII character to be deposited into each byte.

EXAMPLES: FILL 1200 12FF 0 Fills every byte between $1200 & $12FF with $00.
FILL 1000 3FFFF AA Fills all of Datamover user memory with $AA.
FILL 1000 1000+.99 '"' Fills 100 (decimal) bytes starting at $1000 with $22 (an ASCII ").

Notes: 1. As each byte is deposited in memory, it is read back to verify. If the expected value is not read back, the command is aborted at that point.

2.3.9 GET

PURPOSE: To load a Datamover memory image from a disk file.

SYNTAX: GET <file> [:<drive>] [=<dest>...]

ARGUMENTS: <file> = Desired file name to be loaded into memory. The default extension is .6.
<drive> = Drive number, 0-3. Defaults to default drive, usually 0.
<dest> = Destination starting address for load to be used in lieu of the <from> address which was specified when the file was saved.

EXAMPLES: GET MYPROG Loads the file called MYPROG.6 into Datamover memory. It will be loaded into the same addresses from which it was saved. The 68000 PC will be set to the entrypoint specified when the file was saved.

GET OLDPROG.X:1 =1700 =1B00 Will load the file OLDPROG.X from drive 1 into memory. The first block will be loaded starting at $1700 and the second block (if it exists) will be loaded starting at $1B00 regardless of the address specified when the file was saved. Any additional blocks (should they exist) will be loaded into the addresses from which they were saved.

NOTES: 1. The file to be loaded must be a Datamover loadable format file. This means that it must have been created by SAVEing it under DMXMON or by the DMXASM assembler. This is not the same format as a 6502 loadable file. See section 5.3.

2. The file may consist of several non-contiguous blocks of memory, all of which will be loaded. See the SAVE command for details.

3. Naturally, the GET command with <dest> specified does not relocate any machine language code; it merely loads the memory image at a different location. If the 68000 code loaded was written to be position independent, it should run at the new location.

2.3.10 GETLOC

PURPOSE: To display the load addresses and entry point of a Datamover loadable file.

SYNTAX: GETLOC <file> [:<drive>]

ARGUMENTS: <file> = Desired file name. The default extension is .6.
<drive> = Drive number, 0-3. Defaults to default drive, normally 0.

EXAMPLES: GETLOC PROG1 Will display the load addresses and entry point of the file PROG1.6.

GETLOC MATH68K.Q:1 Will display the load addresses and entry point of the file MATH68K.Q on drive number 1.

The typical display format of the data is as follows:

```
MATH68K.Q:1=001078 001000 002467
           012000 013FFF
```

where the value immediately to the right of the = is the entry point address. The next value is the beginning address of the first block and the third value is the last address of the first block. If additional blocks are present, the first and last addresses are listed on additional lines.

NOTES: 1. If the file specified was not created by a DMXMON SAVE command or by the DMXASM assembler, it is probably not in Datamover load format and an error message will result.

2. When using GETLOC to determine memory usage by a program, remember that programs may use additional scratch RAM other than that actually loaded.

2.3.11 GO

PURPOSE: To begin execution of a 68000 machine language program in Datamover memory.

SYNTAX: GO [<from>]

ARGUMENTS: <from> = Desired starting address. Defaults to current value of the program counter (as displayed by the REG command).

EXAMPLES: GO Begins execution at the current value of PC. Note that a previous GET command will set PC to the entry point of the file loaded.
GO 1000 Begins execution at $001000.

NOTES: 1. Upon entry to the program, the registers will be set as displayed (or defined) by the REG command except for the system byte of the status register which will be forced to $27 (supervisor mode, trace off, interrupts disabled). If not previously defined, the registers will have unknown contents. When the program is entered the stack pointer will be set to $0004FC and can go as low as $0002C0.

2. The program is actually entered by signalling the 68000 portion of DMXMON to set the registers and execute a JSR to enter the program. DMXMON then enters its execution mode.

3. If the program terminates with an RTS to the top level, the registers (except PC) will be resaved and DMXMON will enter its console mode. This is useful for debugging subroutines since the REG and GO commands can be used to set up and enter the subroutine. The REG command can then be used to ascertain the results.

2.3.12 GROUP

PURPOSE: To set the 6502 Group Select register (see section 2.2.3 in the Datamover hardware manual) to a specified value while executing CODOS pass-through commands.

SYNTAX: GROUP <group #>

ARGUMENTS: <group #> = Group number to be selected.

EXAMPLES: GROUP 0 Selects Datamover memory group 0 ($000000-$01FFFF) to be mapped into banks 2 and 3 of the 6502's address space.

Group 5 Selects Datamover memory group 5 ($0A0000-$0BFFFF) to be mapped into banks 2 and 3 of the 6502's address space (meaningful only if the Datamover's memory is expanded).

NOTES: 1. The group selection has no effect on any DMXMON commands; it only affects certain CODOS commands that deal with banks 2 and 3.

2. Group 0 is initially selected when DMXMON is started.

3. The primary use for group selection is to allow moving data between any part of Datamover memory and 6502 memory. For example, the MTU-130 display memory may be copied into Datamover locations $3C000-$3FBFF by:

```
.GET IMAGE.G                ;READ IMAGE INTO DISPLAY MEMORY
GROUP 1                     ;SELECT $20000-$3FFFF TO BE IN BANKS 2 & 3
.COPY C000:1 FBFF C000:3    ;COPY IMAGE INTO DATAMOVER
```

2.3.13 HUNT

PURPOSE: To search a block of memory for a string of bytes.

```
                          "<char>"
SYNTAX:  HUNT <from> <to>  <value>    ....
                          '<char>'
                             ?
```

ARGUMENTS: <from> = Starting address for the search
<to> = Final address for the search
<char> = An ASCII character
<value> = A numeric value, 0-$FF
Only one wildcard (?) may be used in a string. Maximum string length is 24 characters.

EXAMPLES: HUNT 2000 2400 'DATAMOVER' Will search memory from $2000 through $2400 inclusive and list the addresses of all occurrances of the ASCII string "DATAMOVER". When a match is made of all 9 bytes, the starting address of the matching string is displayed and the search resumes at the next byte.

HUNT 200 4480 'GO' ? "Paul's" 0D Will search memory from $200 through $4480 for GO followed by any single byte followed by Paul's followed by a carriage return.

NOTES: 1. HUNT reports <u>all</u> occurrences of the target byte-string. CTRL/S may be used to temporarily stop the display and CTRL/Q to restart it. CTRL/C will terminate the command.

2. Only <u>one</u> ? wildcard can be used and it cannot be the first byte of the search string (that would be meaningless).

3. A ? <u>inside</u> an ASCII string is not a wildcard and will only match a ? in memory.

2.3.14 NEXT

PURPOSE: To resume execution after a breakpoint, trap, or interrupt or to initiate execution of a 68000 program.

SYNTAX: NEXT [<from>]

ARGUMENTS: <from> = starting address, defaults to value of the program counter

EXAMPLES:
```
NEXT        Will begin execution at the address currently in PC
NEXT 32F6   Will begin execution at $0032F6
```

NOTES: 1. The program is entered via a JMP instruction so that an RTS instruction will return to the address on the stack, not DMXMON.

2. The difference between GO and NEXT is that GO <u>initializes</u> the stack and system status byte and enters the program with a JSR whereas NEXT enters with a JMP with the stack and system status byte preserved. The advantage of NEXT is that it enables a user to resume execution after a breakpoint, trap, or interrupt.

2.3.15 REG

PURPOSE: To display or alter the contents of the 68000's registers.

SYNTAX: REG [<reg. desig.> [=] <value> "<char>" ....]
'<char>'

ARGUMENTS: <reg. desig.> = Register name to be altered: A0-A6, D0-D7, PC, UP, SP, SR (A7 is the same as SP)
<value> = Desired numeric value or expression
<char> = Desired ASCII character

EXAMPLES:
```
REG                          Will display the register contents.
REG A3=23600+.100            Will set address register 3 to $00023664.
REG D0=0 D1='A' UP=2000      Will set data register 0 to zero, data
                             register 1 to $00000041, and the user stack
                             pointer to $002000.
```

NOTES: 1. The = between the register designator and the value is optional.

2. The REG command without arguments displays the user's registers in the format illustrated below:

```
PC=0276B4 (BD7329A4)  SP=000478  UP=03FC88  SR=2705
D0=8736 AF25   D4=274F A720   A2=0000 541C   A4=004A FF45
D1=FFFF FF67   D5=4142 4344   A1=0000 0000   A5=A900 472B
D2=0000 29B5   D6=0002 1743   A2=FFFF F721   A6=0002 0000
D3=28F4 8736   D7=4142 4344   A3=0001 FF76
```

2.3.16 RESAVE

PURPOSE: To resave one or more blocks of Datamover memory on a CODOS file.

SYNTAX: RESAVE <file> [:<drive>] [=<entry>] <from> [=<dest>] <to> ...

ARGUMENTS:
<file> = Desired file name, default extension is .6
<drive> = Desired drive, 0-3, defaults to default drive, usually 0
<entry> = Entry point desired, defaults to <from>
<from> = Starting address for the block of memory
<dest> = Address at which the block is the be loaded on subsequent GET commands, defaults to <from>
<to> = Final address of the memory block

NOTES: 1. RESAVE performs the same function as SAVE except that it is capable of replacing an existing file. See 2.3.18.

2.3.17 RESET

PURPOSE: To immediately halt and reset the 68000.

SYNTAX: RESET

EXAMPLE: RESET Will activate both the Halt and Reset hardware lines on the 68000 thus immediately halting program execution at the end of the current machine cycle. All pending interrupts to and from the 68000 are also cleared. The 68000 portion of DMXMON will also be reloaded and the default settings of all DMXMON parameters will be reset. After reinitialization, the 68000 portion of DMXMON will be restarted.

PURPOSE: To save one or more blocks of Datamover memory on a CODOS file.

SYNTAX: SAVE <file> [:<drive>] [=<entry>] <from> [=<dest>] <to> ...

ARGUMENTS: <file> = Desired file name, default extension is .6
<drive> = Desired drive, 0-3, defaults to default drive, usually 0
<entry> = Entry point desired, defaults to <from>
<from> = Starting address for the block of memory
<dest> = Address at which the block is the be loaded on subsequent GET commands, defaults to <from>
<to> = Final address of the memory block

EXAMPLES: SAVE DOIT 2000 2DF0

saves the contents of memory locations $002000 through $002DF0 inclusive on a file called DOIT.6 on drive 0 (by default). Since no optional arguments were sepcified, the entry point will be saved on the file as $002000, the same as the starting address of the block. DOIT is now a DMXMON user-command, so subsequently typing DOIT in response to a DMXMON> prompt will cause the block to be loaded from disk into memory at $2000 and execution begun at $2000. The GET command may also be used to reload the file into Datamover memory without executing it.

SAVE RALPH_PROG.Q:1 =2424 2000 20FE 1340 13A0

saves a file called RALPH_PROG.Q on drive 1. The file contains two memory blocks, the first from $002000 to $0020FE and the second from $001340 to $0013A0. The entry point is $0002424.

SAVE SUBPKG.X 1400=3000 1400+.99

saves 100 decimal bytes of memory on a file called SUBPKG.X, starting at $001400. Since a dest. address was specified, a subsequent GET SUBPKG.X command will cause the memory block to be loaded into addresses $3000 and up instead of the $1400 address from which it was saved.

NOTES: 1. The file is saved in DMXMON loadable format (see appendix) which is different from CODOS loadable format.

2. The existence of the "=" in the command indicates the existence of one of the optional arguments, <entry> or <dest>. Pay careful attention to the position of the arguments.

3. When using <dest>, note that no relocation of any possible address references is made; the block saved is still an exact image of memory. Therefore specifying <dest> may not be a satisfactory method of relocating machine language programs.

4. The entry point does not have to reside inside any of the saved blocks.

5. The number of blocks saved on a single file is limited only by the number of <from> <to> arguments you can fit on the command line.

2.3.19 SET

PURPOSE: To set the value of memory locations.

```
                          "<char>..."
SYNTAX: SET <from> [=]    <value>       ...
                          '<char>..."
```

ARGUMENTS: <from> = Address at which to deposit the first value
<value> = Numeric value to be deposited
<char> = An ASCII character to be deposited

EXAMPLES:

```
SET 12000=1B      stores a $1B into Datamover memory location $012000
SET 2006 "ABC"    stores $41 into $2006, $42 into $2007, and $43 into
                  $2008.
SET 1200 80-.10 "   " 80-.20 '"'    sets location $1200 to $76,
                  $1201-$1203 to $20 (ASCII blank), $1204 to $6C, and
                  $1205 to $22 (ASCII ").
```

NOTES:

1. The = is optional and has no effect on the meaning of the command.
2. As each byte is deposited in memory, it is verified by DMXMON. If reading the byte back from memory results in a bad compare to the value deposited, an error message is issued and the command is aborted.

2.4 USER DEFINED COMMANDS

When a Datamover 68000 program is saved as a file under DMXMON, it effectively becomes a DMXMON "User Defined Command". Simply by typing the program file name in response to a DMXMON> prompt, the user can cause the program to be loaded and executed automatically. Arguments can also follow the program filename on the command line, and if the program is written to do so, it can read and interpret those arguments.

To familiarize yourself with this concept, start up DMXMON (see section 2.1) and then type in the following command exactly as written:

```
SET 1000 4E 41 0 2 2 'THIS IS AN INLINE MESSAGE' 0D 0 4E 75
```

This is a short program using an SVC that will simply print a message on the screen and return when it is run. Now enter:

```
GO 1000
```

The message "THIS IS AN INLINE MESSAGE" should be printed on the screen followed by a DMXMON> prompt on the next line which verifies that the program is correct. Now enter this command to save the program on disk:

```
SAVE INLINE 1000 1021
```

This action turns the simple program into a user defined DMXMON command called INLINE. The actual filename saved (as reported by a FILES command) is INLINE.6. Now enter this command to erase the program from memory:

```
FILL 1000 1021 FF
```

You can verify that the program is no longer there by DUMPing the memory block or even trying the GO 1000 again (you will see the message "***NO USER VECTOR FOR EMULATOR OP CODES***" followed by a register printout if you do try the GO again). Now enter the following to execute your new user defined command:

```
INLINE
```

The program will be very quickly read from disk into memory and executed resulting in the "THIS IS AN INLINE MESSAGE" display. Note that user defined commands can be executed from a job file just like any other DMXMON command.

2.5 CODOS PASS-THROUGH COMMANDS

In addition to the Datamover specific DMXMON commands described in section 2.3, a number of frequently used CODOS commands are recognized as such and "passed through" to CODOS for execution. The command function and syntax is exactly as described in the CODOS manual. The list of automatic pass-through commands is given below:

COMMAND	FUNCTION
ASSIGN	Assign a channel to a file or device
BEGINOF	Position a channel to beginning of data
BOOT	Exit DMXMON and reload CODOS operating system
CLOSE	Flush buffers and allow removal of a diskette
COPYF	Copy specified files
DATE	Set the date
DELETE	Delete specified files immediately
DIR	Display full name and size of specified files
DISK	Display remaining space and VSN of open drives
DO	Read commands from a file
DRIVE	Set default disk drive
ENDOF	Position a channel to end of data
FILES	Display all filenames on a specified drive
FORMAT	Erase and prepare a new disk
FREE	Unassign a channel and close its associated file
KILL	Delete specified files with operator verification
LOCK	Write protect specified file
MSG	Display a message on the console
ONKEY	Define a function key legend and substitution string
OPEN	Allow access to disk in specified drive
PROTECT	Disallow alteration of memory used by CODOS
RENAME	Change the name of an existing file
TYPE	Display contents of a text file
UNLOCK	Allow write or delete of specified file
UNPROTECT	Allow alteration of memory used by CODOS
WAIT	Wait for specified time and continue

There are additional CODOS commands that have the same name as DMXMON commands such as DUMP, SAVE, SET, etc. The difference is that the CODOS command refers to 6502 memory while the DMXMON command of the same name refers to Datamover memory. You may force one of these ambiguous commands to be passed through to CODOS by preceeding the command name with a period. Thus the command:

```
SET 5000 12 34 56 78
```

will write into Datamover memory at 5000 (hex) whereas the command:

```
.SET 5000 12 34 56 78
```

will write into 6502 memory just as if it were typed (less the .) in response to a CODOS> prompt.

CODOS utility commands, 6502 user defined commands, and others not in the pass-through list above can also be executed from DMXMON by preceeding their name with a period. The only restriction in such usage is that they not overlay any memory used by DMXMON which is approximately $0054-00AF in page zero and $0700-$4A00 in main memory. See the memory map in the Appendix for exact DMXMON memory usage.

3. EXECUTION MODE SUPERVISOR CALLS

One of the primary functions of DMXMON is to provide a structured method whereby 68000 programs may access all of the software facilities of CODOS and all of the hardware facilities of the MTU-130. Such an interface mechanism is necessary because the Datamover board itself has no direct hardware access to any of these facilities. The SuperVisor Call (SVC) method chosen makes the fact that such I/O requests are processed through the 6502 transparent to 68000 programs.

3.1 THE SVC MECHANISM

The word SVC is a mnemonic for SuperVisor Call. An SVC is effectively a special kind of subroutine call where the routine being called is specified by a function number rather than by a memory address. This is a very flexible method of identification because the actual subroutines can be moved around in memory without affecting the function numbers.

The DMXMON SVC mechanism is very similar to the CODOS SVC mechanism. Instead of a 6502 BRK instruction followed by a one byte function number, a 68000 TRAP instruction followed by a two byte function number is used. Unlike the 6502 however, the 68000 has 16 different TRAP instructions, distinguished by the low 4 bits in the TRAP op code word. DMXMON uses 4 of these traps (0-3) for SVCs and other functions as outlined below:

```
TRAP 0   Used for DMXMON breakpoints.
TRAP 1   Used for the start-wait-complete form of SVCs.
TRAP 2   Used for the start form of SVCs.
TRAP 3   Used to wait-complete a previously started SVC.
```

For most programming, the start-wait-complete SVC form (TRAP 1) is appropriate. When the 68000 programmer needs to perform an I/O function, he codes the following:

```
<previous instruction>
TRAP   1
DC.W   <function #>
<next instruction>
```

When program execution encounters the SVC, control is passed to the DMXMON SVC processor which interprets the function number, performs the I/O operation with the help of the 6502, and then returns to the instruction following the function number. Many SVCs require arguments to be passed in the A and D registers and will return arguments in these registers as well. Some will return yes/no information in the carry flag. SVCs will always preserve the machine registers not involved in argument passing including both stack pointers and both bytes of the status register.

In the SVC descriptions, the size of arguments both passed and returned is given as either byte (.B), word (.W), or long word (.L). For passed arguments, a size designator of .W or .B indicates that the remaining 1 or 3 bytes of the register respectively are ignored. For returned arguments, a .W or .B indicates that the remaining 1 or 3 bytes of the register will be undefined.

With the start-wait-complete form of SVCs, the user's 68000 program is idle during the entire I/O operation. Computation can overlap SVC I/O however by using the techniques described in section 4.3.

In the following sections, each SVC function number is given a mnemonic. It is recommended that the mnemonics be used in your programming so that SVC references will appear on the assembly cross reference listing. Equates for all of the SVC mnemonics may be found in the file called SVC68000.A.

3.2 CODOS DERIVED SVCS

The following SVCs are analogous to CODOS SVCs of the same function numbers. All of the important CODOS functions are provided but number scanning and the 16 bit pseudo processor are not since these functions are trivial to program with the 68000's more powerful instruction set.

ID #	MNEMONIC	FUNCTION
00	SVCRTS	Display registers and return to DMXMON console mode
01	SVCRET	Return to DMXMON console mode
02	SVCOUM	Output an inline message over a channel
03	SVCINB	Input a byte from a channel
04	SVCOUB	Output a byte to a channel
05	SVCINL	Input a line of text from a channel
06	SVCOUL	Output a line of text to a channel
07	SVCOUS	Output a string of text to a channel
12	SVCDFB	Get location of default I/O line buffers and arguments
13	SVCMON	Execute any DMXMON monitor command
14	SVCQCA	Query the assignment of a specified channel
15	SVCINR	Read a record from a channel
16	SVCOUR	Write a record to a channel
17	SVCBOF	Set the file position pointer to beginning-of-data
18	SVCEOF	Set the file position pointer to end-of-data
19	SVCPSF	Specify the file position randomly
20	SVCQFP	Determine the position of a file
21	SVCASS	Assign a channel to a device or a file
22	SVCFRE	Free a specified channel
23	SVCTNC	Truncate a file at the present file position
28	SVCQVN	Query the version number of DMXMON and CODOS
29	SVCQFS	Scan a file or device name to prepare for assignment
30	SVCDAT	Obtain the current 9 character date
31	SVCTIM	Obtain the current 8 character time
32	SVCTOM	Obtain the highest usable Datamover memory address

3.2.1 SVCRTS 0 = $00

PURPOSE: Display register contents and return to DMXMON console mode.

ARGUMENTS: None

DESCRIPTION: SVCRTS returns control to the DMXMON monitor with a display of the register contents. It is functionally equivalent to SVCRET except that the registers are displayed first.

EXAMPLE:

```
....
TRAP  1     ; SVC 2 = PRINT REGISTERS & RETURN TO DMXMON
DC.W  0
....
```

3.2.2 SVCRET 1 = $01

PURPOSE: Return to DMXMON console mode.

ARGUMENTS: None

DESCRIPTION: SVCRET returns control to the DMXMON monitor which then enters console mode and waits for a command from the operator.

```
EXAMPLE:    ....
            TRAP  1     ; SVC 1 = RETURN TO DMXMON
            DC.W  1
            ....
```

3.2.3 SVCOUM 2 = $02

PURPOSE: Output an inline message over a channel.

ARGUMENTS: First byte after SVC = desired channel number.
Second through Nth byte = desired ASCII message text terminated by a zero byte ($00).

DESCRIPTION: SVCOUM can be used to display a message at any point in a program. It does not affect any registers. The message may be any length up to 64K characters and can contain any byte including unprintable characters except NUL ($00) which is the message terminator. Control will be returned to the instruction immediately following the 0-byte terminator. The channel specified must be assigned to a valid device or file.

```
EXAMPLE:        ....
                CMP.W   D3,HILIM    CHECK D3 AGAINST LIMIT
                BLE     ISOK        BRANCH IF LESS THAN LIMIT
                TRAP    1           SVC 2 = OUTPUT INLINE MESSAGE
                DC.W    2
                DC.B    2           ON CHANNEL 2
                DC.B    $0D,'ERROR-UPPER LIMIT EXCEEDED',0
                JMP     GETN        GO GET ANOTHER NUMBER
            ISOK ....
```

This program segment will compare the contents of D3 with HILIM and print the message "ERROR-UPPER LIMIT EXCEEDED" if D3 is greater than HILIM.

NOTES: 1. The message will always be displayed starting at the present position. If the message should start with a new line, then the carriage return should be explicitly included as in the example.

2. DMXMON always resumes program execution at the next even addressed byte after the message (68000 hardware requirement). Most assemblers will also insure that the next instruction after the message will start at an even address. When hand assembling code, you are responsible for skipping an extra byte if the message is an odd number of bytes long.

3.2.4 SVCINB 3 = $03

PURPOSE: Input a byte from a channel.

ARGUMENTS PASSED: D1.B = Channel number

ARGUMENTS RETURNED: D0.B = Byte received from the channel
CY = Set means that channel was at end of file

DESCRIPTION: SVCINB inputs a single byte from a selected channel, which must be assigned to a valid device or file. The value of the byte returned can be anything, including control characters ($00 to $FF), if the selected channel is assigned to a file. If assigned to a normal, character-oriented input device, such as the keyboard, then a CTRL/Z (ASCII SUB, $1A) will be interpreted as end-of-file. For files, end-of-file is true only when no more bytes can be read from the file. It is the programmer's responsibility to check the status of the Carry after every SVCINB to ensure that end-of-file was not reached, The D0 register is not meaningfully returned if the Carry is set.

EXAMPLE:
```
....
MOVE.B  #5,D1   SELECT CHANNEL 5
TRAP    1       SVC 3 = INPUT BYTE
DC.W    3
BCS     EOF     BRANCH IF END OF FILE
CMP.B   D0,#'C' WAS INPUT CHARACTER A 'C'?
....
```

This program segment inputs a character from the file or device assigned to channel 5 and checks to see if it was an ASCII "C".

NOTES: 1. The remaining flags (other than CY) are not changed.

2. Any value byte can be input including $00, $08, $7F, $FF, etc. No editing characters are recognized.

3. For applications requiring high-speed disk input of large amounts of data, SVCINR (15) is preferred to SVCINB. Since the SVC processor is called for every byte of input using SVCINB, the overhead involved limits throughput to less than 1000 bytes per second. SVCINR is capable of throughput in excess of 15,000 bytes per second.

3.2.5 SVCOUB 4 = $04

PURPOSE: Output a byte to a channel.

ARGUMENTS PASSED: D0.B = Byte to be output
D1.B = Channel number

ARGUMENTS RETURNED: CY = 1 if at end-of-file after output operation

DESCRIPTION: SVCOUB outputs the byte in D0 over the channel specified by D1. The channel must be assigned to a valid file or device. Although there is no need to do so, application programs may wish to test the Carry flag after SVCOUB to distinguish whether the character written was the last character of the file or was re-written over some other part of the file. If the channel is assigned to a device instead of a file, the Carry will always be returned set.

EXAMPLE:

```
....
MOVE.B  #2,D1    SELECT CHANNEL 2
MOVE.B  #$07,D0  GET CHARACTER TO OUTPUT
TRAP    1        SVC 4 = OUTPUT BYTE
DC.W    4
BCS     EOF      BRANCH IF WRITTEN AT END OF FILE
....
```

This program segment outputs $07 over channel 2. Note that $07 is not the character "7" but simply a byte with value 7. In channel 2 is assigned to the MTU-130 console display (the usual case), this will sound a short beep through the speaker since $07 is the ASCII BEL control character.

NOTES: 1. The value $00 (NUL) can be output using SVCOUB, as can any other possible 8 bit code.

2. For applications requiring high-speed disk output of large amounts of data, SVCOUR (16) is preferred to SVCOUB. Since the SVC processor is called for every byte of output using SVCOUB, the overhead involved limits throughput to less than 1000 bytes per second. SVCOUR is capable of throughput in excess of 15,000 bytes per second.

3. After you have finished writing to a file, it is good practice to FREE the channel. This ensures that the system buffer for that file will be "flushed" to disk and that the directory will be updated. Otherwise, the actual disk contents will not be updated until you CLOSE the disk or change the file position. See SVCFRE (22) for additional information.

3.2.6 SVCINL 5 = $05

PURPOSE: Input a line of text from a channel.

ARGUMENTS PASSED: D1.B = Channel number

ARGUMENTS RETURNED: D0.B = number of characters in the line
CY = 1 if end-of-file was encountered immediately
Line returned in input line buffer (pointer at $000260)

DESCRIPTION: SVCINL inputs a line of text from the file or device assigned to the specified channel. The text will be deposited in a buffer whose address is specified in location $000260. The default buffer location is $000500, see section 3.2.9 for details. The line of text will be terminated by a CR ($0D). After the SVC is processed, the Carry will be set only if no characters could be read from the channel because end of file was encountered. The D0 register will contain a character count for the input line. This count does not include the $0D terminator. End of line is defined as the first carriage return ($0D) encountered.

EXAMPLE:

```
....
MOVE.L  #MYIBUF,$260  SET UP ADDRESS OF INPUT BUFFER
MOVE.B  #1,D1   SELECT CHANNEL 1
TRAP    1       SVC 5 = INPUT LINE
DC.W    5
BCS     EOF     BRANCH IF END-OF-FILE
TST.B   D0      TEST FOR NULL LINE
BEQ     ISNULL  JUMP OUT IF SO
....
```

This program segment inputs a line of text from channel 1 (normally assigned to the console) and places it in MYIBUF. It also tests for end-of-file and null line (only a carriage return).

NOTES: 1. The line input cannot be longer than 192 characters (DMXMON uses CODOS's default input line buffer). If SVCINL attempts to input a line with more than 192 characters, it will automatically add a carriage return after the 192nd character and return.

2. When SVCINL is used to read from a channel assigned to the Console, all normal system editing characters (such as insert/delete, rubout, CTRL/B. etc.) will be in effect. If an attempt is made to input more than 192 characters from the Console, a short beep will be sounded and no more characters will be accepted for insertion. This affords the user the chance to backup and change the line, perhaps to make it fit in the 192 character limit.

3. No editing characters are recognized when reading from any source other than the Console.

4. Within DMXMON is a 256 byte area that may be used for an input line buffer. See the description for SVCDFB (12) for details.

5. When maximum throughput is essential for reading disk files, you may wish to use SVCINR (15) instead of SVCINL. If you use SVCINR to read a large block from disk and then remove lines from the buffer individually as needed, throughput will normally be substantially enhanced.

3.2.7 SVCOUL 6 = $06

PURPOSE: Output a line of text to a channel.

ARGUMENTS PASSED: D0.B = Number of bytes in line
D1.B = Channel number
(000264) = address of buffer containing the text

ARGUMENTS RETURNED: None

DESCRIPTION: SVCOUL outputs a line of text over a channel which must be assigned to a valid file or device. Location $000264 must contain a pointer to a buffer containing the text to be sent. The default output line buffer address is $000600, see section 3.2.9 for details. D0 must hold the number of characters to be sent, not including a carriage return which will be added automatically.

```
EXAMPLE:    ....
            MOVE.L  #MYOBUF,$264  SET UP ADDRESS OF BUFFER
            MOVE.B  LLNG,D0 GET LINE LENGTH
            MOVE.B  #2,D1   SELECT CHANNEL 2
            TRAP    1       SVC 6 = OUTPUT LINE
            DC.W    6
            ....
```

This program segment will output a line of text in MYOBUF of length (LLNG) to channel 2 (normally assigned to the display). A carriage return will be appended to the text output from the buffer.

NOTES: 1. The character count must be passed in the low byte of D0. This is normally convenient because D0 can be used as an index register that is incremented after each character is assembled into the buffer.

2. Within DMXMON is a 256 byte area that may be used for an output line buffer. See the description for SVCDFB (12) for details.

3. The difference between SVCOUL and SVCOUS (7) is that SVCOUL adds a carriage return after the line is output and SVCOUS does not.

3.2.8 SVCOUS 7 = $07

PURPOSE: Output a string of text to a channel.

ARGUMENTS PASSED: D0.B = Number of bytes in string
D1.B = Channel number
(000264) = Address of buffer containing the text

ARGUMENTS RETURNED: None

DESCRIPTION: SVCOUS outputs a string of text over a channel which must be assigned to a valid file or device. Location $000264 must contain a pointer to a buffer containing the text to be sent. The default output buffer is at $000600, see section 3.2.9 for details. D0 must hold the number of characters to be sent.

```
EXAMPLE:    ....
            MOVE.L  #MYOBUF,$264  SET UP ADDRESS OF BUFFER
            MOVE.B  SLNG,D0 GET STRING LENGTH
            MOVE.B  #2,D1   SELECT CHANNEL 2
            TRAP    1       SVC 7 = OUTPUT STRING
            DC.W    7
            ....
```

This program segment will output a string of text in MYOBUF of length (SLNG) to channel 2 (normally assigned to the display). The string will be output exactly as it appears in the buffer with no end-of-line character added by the system.

NOTES: 1. The character count must be passed in the low byte of D0. This is normally convenient because D0 can be used as an index register that is incremented after each character is assembled into the buffer.

2. Within DMXMON is a 256 byte area that may be used for an output line buffer. See the description for SVCDFB (12) for details.

3. The difference between SVCOUS and SVCOUL (6) is that SVCOUL adds a carriage return after the line is output and SVCOUS does not. Naturally carriage returns may be embedded in the string itself if desired.

3.2.9 SVCDFB 12 = $0C

PURPOSE: Obtain location of system input line buffer, output line buffer, and arguments passed to user-defined DMXMON command.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: $000260 = address of input line buffer
$000264 = address of output line buffer
Arguments copied into input line buffer

DESCRIPTION: SVCDFB installs the addresses of default DMXMON-supplied input and output line buffers into $000260 and $000264 respectively. Each of these buffers is 256 bytes in length and is contained entirely in DMXMON memory between $000000 and $000FFF. SVCDFB also copies any arguments that may follow the DMXMON command that initiated execution of the program issuing SVCDFB. The arguments, if any, are are copied without leading blanks before the first argument. If there are no arguments, the first byte in the input line buffer will be a carriage return ($0D).

EXAMPLE:

```
....
TRAP    1         SVC 12 = ESTABLISH DEFAULT BUFFERS
DC.W    12
MOVEA.L $260,A0   TEST IF ANY ARGUMENTS PASSED
CMP.B   (A0),#$0D
BEQ     NONE      JUMP IF NONE
CMP.B   (A0),#'Y' TEST IF ARGUMENT IS "Y"
BEQ     ISY       JUMP IF SO
CMP.B   (A0),#'N' TEST IF ARGUMENT IS "N"
BEQ     ISN       JUMP IF SO
TRAP    1         OTHERWISE, ERROR
DC.W    2
DC.B    2,"ILLEGAL ARGUMENT",13,0
....
```

This program segment will establish addresses of the default buffers in locations $260 and $264 and copy the command arguments into the input line buffer whose address is in location $260. If there are no arguments, a jump to NONE is taken. If the first argument is "Y" a jump to ISY is taken, if it is "N" a jump to ISN is taken and if it is something else, an error message is printed. For example, if the above program segment had been saved on a file and initiated by the DMXMON command:

```
MYCOMD   N
```

then the character "N" would be first in the input line buffer and the program would jump to ISN.

NOTES: 1. Any number of arguments using any syntax desired may be passed from the command line to the input line buffer. Copying starts with the first non-blank after the command name and ends when a carriage return is encountered.

2. The address of the input line buffer that will be deposited into location $260 is $500 and the address of the output line buffer is $600.

3. DMXMON does not use the default input and output line buffers for any purpose when in command mode.

3.2.10 SVCMON 13 = $0D

PURPOSE: To execute any DMXMON monitor command.

ARGUMENTS PASSED: A0 = Address of command string to execute

ARGUMENTS RETURNED: None except register contents and memory locations that may be changed by the command's own execution.

DESCRIPTION: SVCMON (13) is the most powerful of all SVCs provided. Creatively used, it can give tremendous leverage to an application program. Simply stated, SVCMON calls the DMXMON monitor as a subroutine, with the command read from memory instead of channel 1 (normally the console). Each invocation of SVCMON will execute one DMXMON monitor command and then return to the invoking program in the normal manner. Thus a program can easily OPEN and CLOSE disk drives, FILL or COPY memory, GET, SAVE, or TYPE files, etc. Utilities and CODOS user-defined commands (6502 code) may also be executed in the usual manner. To use SVCMON, the desired command must exist as an ASCII string in memory terminated by a carriage return ($0D) and address register 0 must contain the address of the start of this command string.

EXAMPLE:

```
        ....
        MOVEA.L #DCMMD,A0  SET POINTER TO "DISK" COMMAND
        TRAP    1          SVC 13 = EXECUTE THE "DISK" COMMAND WHICH WILL
        DC.W    13         DISPLAY DISK VSN'S AND SPACE REMAINING
        TRAP    1          FOLLOW WITH QUESTION TO OPERATOR
        DC.W    SVCMSG
        DC.B    2,$0D      PRINT ON CHANNEL 2=DISPLAY
        DC.B    'ARE THESE THE RIGHT DISKS AND IS THAT ENOUGH SPACE? ',0
        TRAP    1          READ A BYTE FROM CHANNEL 1=KEYBOARD
        DC.W    SVCINB
        CMP     D0,#'Y' TEST IF "Y" REPLY
        ....
DCMMD   DC.B    'DISK',$0D
```

This program segment will first execute a "DISK" command which will display the volume serial numbers and remaining capacity of all open disks. It then uses SVCMSG (2) to print a question to the operator and SVCINB to read a single character reply which then determines the program's future action.

NOTES: 1. The command executed may be either a DMXMON command, a CODOS pass-through command (either automatic or forced with a period prefix), or a user-defined CODOS command (6502 code). A DMXMON user defined command (68000 code) should not be attempted.

2. Since the return path to the invoking program is stored on the system stack, the command executed must not redefine the system stack pointer except by normal usage of balanced JSR, RTS, pushes, pops, etc.

3.2.11 SVCQCA 14 = $0E

PURPOSE: To query the assignment for a selected channel.

ARGUMENTS PASSED: D1.B = Channel number

ARGUMENTS RETURNED: D0.B = Disk drive number if returned as 0-3; otherwise, returned as the single character device name.
CY = 1 if the channel is assigned, =0 if free.

DESCRIPTION: SVCQCA (14) enables a program to determine if a specified I/O channel is assigned or is available. If it is assigned, then the device or drive it is assigned to can also be determined.

EXAMPLE:
```
....
MOVE.B  #5,D1   QUERY CHANNEL 5
TRAP    1       SVC 14 = QUERY CHANNEL ASSIGNMENT
DC.W    14
BCC     NOTASG  JUMP IF NOT ASSIGNED
CMP.B   D0,#3
BGT     ISDEV   JUMP IF ASSIGNED TO A DEVICE
....            ELSE ASSIGNED TO A FILE
```

This program segment determines the status of channel 5 and jumps to NOTASG if it is not assigned, jumps to ISDEV if assigned to a device, and continues if assigned to a file.

3.2.12 SVCINR 15 = $0F

PURPOSE: To read a record from a channel.

ARGUMENTS PASSED: D0.L = Number of bytes to read
D1.B = Channel number
A0.L = Starting memory address to receive the record

ARGUMENTS RETURNED: D0.L = Number of bytes actually read
A0.L = Address of last byte read, plus one
CY = 1 if end-of-file encountered before any bytes could be read

DESCRIPTION: SVCINR reads a block of bytes from a channel. Any size of block may be read and the size need bear no relation to the structure of the data read.

EXAMPLE:
```
....
MOVE.L  #40000,D0  SET RECORD LENGTH = 40000
MOVE.B  #5,D1      SET TO READ FROM CHANNEL 5
MOVEA.L #BIGBUF,A0 SET RECORD MEMORY ADDRESS
TRAP    1          SVC 15 = READ RECORD
DC.W    15
BCS     ISEOF      JUMP OUT IF FILE ALL READ
CMP.L   #40000,D0  CHECK IF ALL ASKED FOR WAS READ
BNE     NOTALL
....
```

This program segment reads 40,000 bytes from channel 5 into a buffer starting at BIGBUF. After the read, it jumps to ISEOF if no bytes were read because of end-of-file and jumps to NOTALL if at least 1 but fewer than 40,000 bytes were successfully read.

NOTES: 1. If the specified channel is assigned to a file, then reading begins at the current file position and continues until the specified number of bytes are read or end-of-file is encountered. Any type of data bytes may be read; there are no reserved end-of-record or end-of-file characters.

2. If the channel is assigned to a device (not a file), then reading continues until the specified number of bytes are read or the end-of-file character (ASCII SUB = $1A = CTRL/Z) is read. The CTRL/Z is not returned as part of the record in memory.

3. If the Carry flag is returned set then no bytes at all could be read from the channel because end-of-file was encountered immediately.

4. If the carry flag is returned clear than at least 1 byte was read before end-of-file. The actual number of bytes read is returned in D0. If D0 contains a smaller count than was specified but the carry remained clear, it indicates that end-of-file was encountered during the read operation.

5. When reading large amounts of data from a file, using large records will significantly improve the reading speed. For example, if a program is to read 1000 80-character records, the obvious way to do it is to use a loop which invokes SVCINR 1000 times with D0 set to 80. However a significant speed improvement can be realized by instead using, say, 100 SVCs of 800 bytes each. By using large records it is possible to read in excess of 15,000 bytes per second continuous throughput from a file.

6. The only practical limits to the record size is available memory (256K on the standard Datamover) and the capacity of a single diskette (1Mbyte for 8" floppies).

3.2.13 SVCOUR 16 = $10

PURPOSE: To write a record to a channel.

ARGUMENTS PASSED: D0.L = Number of bytes to write
D1.B = Channel number
A0.L = Starting address in memory of record to write

ARGUMENTS RETURNED: CY = 1 if the channel was positioned at end-of-file after completing the write.

DESCRIPTION: SVCOUR writes a block of bytes to a channel. Any size of block may be written and the size need bear no relation to the structure of the data written.

EXAMPLE:

```
....
MOVE.L  #5000,D0    SET RECORD LENGTH = 5000
MOVE.B  #6,D1       SET TO WRITE TO CHANNEL 6
MOVEA.L #MIDBUF,A0  SET RECORD MEMORY ADDRESS
TRAP    1           SVC 16 = WRITE RECORD
DC.W    16
BCS     NEWEOF      JUMP OUT IF END OF FILE WAS EXTENDED
....
```

This program segment writes a record on channel 6. The record to be written is 5000 decimal bytes long and starts at MIDBUF in memory. After the write, it jumps to NEWEOF is the write extended the size of the file assigned to channel 6.

NOTES: 1. If the specified channel is assigned to a file, then writing begins at the current file position. Any type of data bytes may be written; no special end-of-record characters will be written by the system. If the Carry is returned clear, it indicates that the file was not positioned to end-of-file on the completion of the write operation (therefore part of the file must have been overwritten).

2. If the channel is assigned to a device (not a file), then writing continues until the specified number of bytes have been output. The Carry flag is always returned set when writing to a device.

3. Using large records will improve writing speed. For example, writing 100 records of 80 bytes each takes longer than writing 10 records of 800 bytes each. Continuous output to disk in excess of 15,000 bytes per second is possible by using large records.

4. After you have finished writing to a file, it is a good practice to FREE the channel. This insures that the system buffer for that file will be "flushed" to disk and that the directory will be updated. Otherwise, the actual disk contents will not be updated until you CLOSE the disk or change the file position.

5. The only practical limits to the record size is available memory (256K on the standard Datamover) and the capacity of a single diskette (1Mbyte for 8" floppies).

3.2.14 SVCBOF 17 = $11

PURPOSE: To set the file position for a channel to beginning-of-data.

ARGUMENTS PASSED: D1.B = Channel number

ARGUMENTS RETURNED: None

DESCRIPTION: After executing SVCBOF (17), a subsequent read or write operation will access the first data byte of the file assigned to the specified channel.

EXAMPLE:
```
....
MOVE.B  #4,D1       SET TO POSITION CHANNEL 4
TRAP    1           SVC 17 = POSITION TO BOF
DC.W    17          "REWIND" THE FILE
....
```

This program segment positions the file assigned to channel 4 to beginning-of-data.

NOTES: 1. If the selected channel is assigned to a device instead of a file, no action takes place.

2. A file is always initially positioned to beginning-of-data when it is assigned.

3. Executing SVCBOF will always result in a physical disk access, even if the file is already positioned at beginning-of-data.

3.2.15 SVCEOF 18 = $12

PURPOSE: To set the file position for a channel to end-of-file.

ARGUMENTS PASSED: D1.B = Channel number

ARGUMENTS RETURNED: None

DESCRIPTION: SVCEOF (18) positions the file assigned to the specified channel to end-of-file. A subsequent write operation would therefore append new data to the file.

```
EXAMPLE:    ....
            MOVE.B  #8,D1       SET TO POSITION CHANNEL 8
            TRAP    1           SVC 17 = POSITION TO EOF
            DC.W    18
            ....
```

This program segment positions the file assigned to channel 8 to end-of-file.

NOTES: 1. If the selected channel is assigned to a device instead of a file, no action takes place.

2. A file is always initially positioned to beginning-of-data when it is assigned.

3. Executing SVCEOF will always result in a physical disk access, even if the file is already positioned at end-of-file.

3.2.16 SVCPSF 19 = $13

PURPOSE: To specify the file position for a channel.

ARGUMENTS PASSED: D0.L = Desired file position
D1.B = Channel number to position

ARGUMENTS RETURNED: None

DESCRIPTION: SVCPSF (19) positions the file assigned to the specified channel to the position specified by D0.

```
EXAMPLE:    ....
            MOVE.L  #7518,D0    SET POSITION TO ACCESS 7519TH BYTE OF FILE
            MOVE.B  #6,D1       SET TO POSITION CHANNEL 6
            TRAP    1           SVC 19 = POSITION FILE RANDOMLY
            DC.W    19
            MOVE.B  #' ',D0     BLANK OUT THAT POSITION USING
            TRAP    1           AN OUTPUT BYTE SVC
            DC.W    SVCOUB
            ....
```

This program segment positions the file assigned to channel 6 to the 7518th byte and changes it to a blank.

NOTES: 1. If the selected channel is assigned to a device instead of a file, no action takes place.

2. A file is always initially positioned to beginning-of-data when it is assigned.

3. Executing SVCPSF will always result in a physical disk access, even if the file is already positioned at location specified by D0.

4. If the position specified by D0 is beyond the current end-of-file, the file will be positioned to the current end-of-file.
5. The beginning of a file is position 0. The next byte read when the position is 0 will be the first byte of the file. Position 1 represents the position after the first byte in the file. The next byte read when the position is 1 will be second byte, etc.

3.2.17 SVCQFP 20 = $14

PURPOSE: To determine the position of a file assigned to a channel.

ARGUMENTS PASSED: D1.B = Channel number to query

ARGUMENTS RETURNED: D0.L = Current file position

DESCRIPTION: SVCQFP (20) returns the present file position for the specified channel in D0.

EXAMPLE:

```
....
MOVE.B  #6,D1       SET TO QUERY CHANNEL 6
TRAP    1           SVC 20 = QUERY FILE POSITION
DC.W    20
CMP.L   #65536,D0   CHECK IF BEYOND 64K
BGE     PG1         JUMP IF SO
....
```

This program segment determines the position of the file assigned to channel 6. If it is beyond 64K, then a branch to PG1 is taken.

NOTES: 1. If the selected channel is assigned to a device instead of a file, then D0 is always returned as $00000000.

3.2.18 SVCASS 21 = $15

PURPOSE: To assign a channel to a device or a file.

ARGUMENTS PASSED: D0.B = Drive number or device name
D1.B = Channel number to assign
A0.L = Address of file name (applies to assignment to a file)

ARGUMENTS RETURNED: D0.B = Byte of status information as follows:
Bit 7 = Old flag. If set then file already exists.
Bit 6 = File flag. If set, then channel assigned to a file, not a device.
Bit 5 = Locked flag. If set, then file is locked (read only).

DESCRIPTION: SVCASS (21) assigns the channel specified by D1 to the device or disk drive specified in D0 and file name specified by a string whose address is in A0.

```
EXAMPLE:    ....
            MOVEA.L #TNAME,A0  POINT A0 TO TEMP FILE NAME
            MOVE.B  #1,D0      PUT FILE ON DRIVE 1
            MOVE.B  #6,D1      SET TO ASSIGN CHANNEL 6
            TRAP    1          SVC 21 = ASSIGN CHANNEL TO A FILE
            DC.W    21
            BTST    #7,D0      TEST IF FILE EXISTS
            BEQ     ISNEW      SKIP AHEAD IF IT DOES NOT EXIST
            ....
      TNAME DC.B    'TEMP.D'   NAME OF TEMPORARY FILE
```

This program segment assigns channel 7 to a file called TEMP.D and jumps to ISNEW if the file had not been created previously.

NOTES: 1. A0 is not used if the channel is assigned to a device (not file).
LEFT=8 REVIND=3

2. The file name can be terminated by any character which is not legal in a file name. The extension may be included or omitted (in which case it defaults to .C).

3. Assigning a channel to a file always positions the file to beginning of data.

4. Assigning a channel which is already assigned automatically frees the old channel assignment before making the new assignment.

5. The maximum length of the file name including the extension is 14 characters.

6. Drive numbers as part of the file name (:<drive>) are not recognized by SVCASS. See SVCQFS (29) for more information on file assignments.

3.2.19 SVCFRE 22 = $16

PURPOSE: To free a channel.

ARGUMENTS PASSED: D1.B = Channel number to free.

ARGUMENTS RETURNED: None

DESCRIPTION: SVCFRE frees the channel specified by D1. If the channel had been assigned to a file, the file buffer will be flushed to the file and the disk directory will be updated.

```
EXAMPLE:    ....
            MOVE.B  #6,D1      FREE CHANNEL 6 AND FLUSH BUFFER
            TRAP    1          SVC 22 = FREE A CHANNEL
            DC.W    22
            ....
```

This program segment frees channel 6, flushes the file's buffer in CODOS, and updates the directory.

NOTES: 1. Freeing a channel which is unassigned results in no action.

2. It is important that programs which write to disk always free the channel when the file is completed. Otherwise, if the disk is removed from the drive by the operator without a CLOSE command, the file could be incomplete.

3.2.20 SVCTNC 23 = $17

PURPOSE: To truncate a file at the present file position.

ARGUMENTS PASSED: D1.B = Channel number to to truncate

ARGUMENTS RETURNED: None

DESCRIPTION: SVCTNC (23) makes the present file position end-of-file. It is normally used to discard the unwanted end part of a file, or to discard the unwanted residual when overwriting an existing file with a shorter file.

```
EXAMPLE:    ....
            MOVE.L  #8192,D0    POSITION FILE TO 8K BYTES
            MOVE.B  #4,D1       FILE ASSIGNED TO CHANNEL 4
            TRAP    1           DO THE POSITION
            DC.W    SVCPSF
            TRAP    1           SVC 23 = TRUNCATE FILE AT THAT POSITION
            DC.W    23
            TRAP    1           FREE THE CHANNEL
            DC.W    SVCFRE
            ....
```

This program segment truncates the file to a maximum of 8K bytes of data. If the file contained less than 8K of data, it would not be changed. If it contained more than 8K of data, the rest of the file would be discarded.

NOTES: 1. If the channel is assigned to a device instead of a file, no action takes place.

3.2.21 SVCQVN 28 = $1C

PURPOSE: To return information about the version of CODOS and DMXMON which is running.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.W = Information as follows:
Bits 12-15 = DMXMON version number
Bits 8-11 = DMXMON revision number
Bits 4-7 = CODOS version number
Bits 0-3 = CODOS revision number

DESCRIPTION: SVCQVN returns information about the particular version of CODOS and DMXMON that is running. The least significant byte identifies the CODOS version and the next byte identifies the DMXMON version. Each byte is split into two hex digits, HL which correspond to the version number, H.L.

```
EXAMPLE:    ....
            TRAP    1           DETERMINE OPERATING SYSTEM VERSION
            DC.W    SVCQVN
            CMP.B   #$1F,D0     TEST IF CODOS VERSION 2.0
            BGT     ISOK        SKIP IF OK
            TRAP    1           CORRECT EFFECTS OF FILE BUG
            DC.W    SVCOUM      PRINT ERROR MESSAGE, NEEDED FEATURES NOT
            ....                AVAILABLE IN EARLIER VERSIONS
```

This program segment determines whether the program environment is is less than version 2.0 for CODOS. If so, it prints an error message because the program uses some features that are not in the earlier versions.

NOTES: 1. This SVC is useful when writing programs that depend on some version dependent feature or undocumented subroutine entrypoint. It allows the program to verify its operating environment if necessary before continuing.

3.2.22 SVCQFS 29 = $1D

PURPOSE: To scan a device or file name:drive in preparation for assignemnt to a channel or to ascertain a file's status.

ARGUMENTS PASSED: A0.L = Address of file or device name character string

ARGUMENTS RETURNED: A0.L = Address of first byte after the name.
CY is set if parsed name was a device name, clear if parsed name is a file name.
D0.B = Information about the name as follows:
For device name (CY set):
Bit 7 set if specified device does not exist
Bits 6-0 contain the ASCII device name (1 byte)
For file name (CY clear):
Bit 7 set if illegal name or drive number
Bit 6 set if specified drive is not open
Bit 5 set if the file already exists
Bit 4 set if the drive is write protected
Bits 3 and 2 are not used
Bits 1 and 0 contain the drive number selected

DESCRIPTION: SVCQFS (29) is a multi-purpose SVC which performs the following activities:

1. It scans and validates an input string of characters specifying a file or device name.

2. It returns status bits so that the application program can gracefully recover from common file/device specification errors without aborting the program.

3. It helps setup the necessary registers for using SVCASS (21).

SVCQFS is normally used to parse a character string which specifies the name of a file or device which is to be assigned to a channel for I/O. SVCQFS scans the string, determines if the name is a device name or a file name, and checks it for legality.

If the name is a legal device name, it checks to see if the specified device exists on the system. For example, a printer may or may not be present. The device name is returned in D0 so that the user may immediately use SVCASS to assign the device to a channel if all is in order.

If the name is a file name, it is checked for legality. An optional drive number may be specified as part of the name if the name is terminated by a colon followed by a digit. Blanks may be present between the colon and the digit. If no drive is given as part of the string, the default drive (usually drive 0) will be assumed. If the file name (and optionally the drive number) are legal, then SVCQFS checks to see if the drive is open. If it is, then the disk is checked for write protection. The directory is then searched for the file name. The status bits of D0 return the results of these operations to the application program. Upon completion of SVCQFS, the application program should verify that the status bits returned reflect the desired conditions. If they do, then all bits of D0 except 0 and 1 should be masked off and SVCASS (21) invoked to assign the file.

```
EXAMPLE:    ....
            TRAP    1           USE STANDARD BUFFERS AND GET ARGUMENTS
            DC.W    SVCDFB
            MOVEA.L $260,A0     GET ADDRESS OF FIRST ARG (=NAME) IN A0
            TRAP    1           PARSE THE NAME
            DC.W    SVCQFS
            MOVEA.L A0,A4       SAVE POINTER TO NEXT ARG FOR LATER
            BTST    #7,D0       BRANCH IF ILLEGAL OR NON-EXISTANT
            BNE     NOGOOD
            BCS     ASGNIT      BRANCH IF DEVICE SPECIFIED
            BTST    #6
            BNE     NOTOPN      BRANCH IF SPECIFIED DRIVE IS NOT OPEN
            BTST    #5
            BEQ     NOSUCH      BRANCH IF NO SUCH FILE IS PRESENT
            ANDI.B  #$03,D0     ELSE DISCARD ALL BUT DRIVE NUMBER
ASGNIT      MOVE.B  #5,D1       SET TO ASSIGN TO CHANNEL 5
            MOVEA.L $260,A0     SET ADDRESS OF NAME IN A0 AGAIN
            TRAP    1           DO THE ASSIGNMENT
            DC.W    SVCASS
            ....
```

This program segment first uses SVCDFB (12) to get the address of the system input buffer and a copy of the argument which is presumably a device or file name. It then uses SVCQFS (29) to parse the first argument encountered. The return arguments from SVCQFS are checked to make sure the file or device exists and is ready to be read. If everything is OK, then control passes to ASGNIT which assigns the file or device name just scanned to channel 5 in preparation for reading. If there is a problem with the syntax of the argument or the file/device name, branches are taken to appropriate error handling routines within the program (not shown).

NOTES: 1. SVCQFS does not indicate whether a file is locked. The assign SVC (SVCASS, 21) however does indicate the locked/unlocked status of a file when assignment of a channel to a file is completed.

2. The total length of the file or device name, including blanks, must not be more than 20 characters.

3.2.23 SVCDAT 30 = $1E

PURPOSE: To obtain the current 9 character date.

ARGUMENTS PASSED: A0.L = Address of buffer to copy the date into

ARGUMENTS RETURNED: A0.L = Passed A0 + 9

DESCRIPTION: SVCDAT is used to obtain the current date as entered during the normal CODOS signon procedure or by execution of the DATE command. The date field will be 9 characters long and is stored in a buffer pointed to by A0. A0 is then incremented automatically by 9 in preparation for additional output into the same buffer.

EXAMPLE:

```
....
TRAP    1
DC.W    SVCOUM     OUTPUT MESSAGE ON CHANNEL 2
DC.B    2,'TODAY IS ',0
MOVEA.L #BUFF,A0   SET BUFFER ADDRESS IN A0
TRAP    1          DETERMINE TODAY'S DATE
DC.W    SVCDAT
MOVEA.L #BUFF,A0   RESET A0
MOVE    #9,D0      SET 9 CHARACTER COUNT
MOVE    #2,D1      SET CHANNEL 2
TRAP    1          PRINT THE DATE AND CR
DC.W    SVCOUL
....
```

This program segment first prints on channel 2 (usually the display) using SVCOUM (2) "TODAY IS ", then reads the 9 character date using SVCDAT into a buffer at BUFF, and then prints the date followed by a carriage return on channel 2 from the buffer using SVCOUL (6).

NOTES: 1. The default date field is "%UNDATED%".

2. The exact format of the date is dependent on what the operator typed in when the system was started. It will typically be either DD-MMM-YY or MM/DD/YY with trailing blanks if the actual date is less than 9 characters.

3.2.24 SVCTIM 31 = $1F

PURPOSE: To obtain the current 8 character time of day.

ARGUMENTS PASSED: A0.L = Address of buffer to copy the time into

ARGUMENTS RETURNED: A0.L = Passed A0 + 8

DESCRIPTION: SVCTIM is used to obtain the current time from the MultI/O board if it is present in the system. The time field will be 8 characters long and is stored in a buffer pointed to by A0. A0 is then incremented automatically by 8 in preparation for additional output into the same buffer.

```
EXAMPLE:    ....
            TRAP    1
            DC.W    SVCMOUM    OUTPUT MESSAGE ON CHANNEL 2
            DC.B    2,'THE DATE AND TIME IS: ',0
            MOVEA.L #BUFF,A0   SET BUFFER ADDRESS IN A0
            TRAP    1          COPY DATE INTO BUFFER
            DC.W    SVCDAT
            MOVE.B  #' ',(A0)+ PUT IN A BLANK SEPARATOR
            TRAP    1          COPY TIME INTO BUFFER
            DC.W    SVCTIM
            MOVEA.L #BUFF,A0   RESET A0
            MOVE    #18,D0     SET 18 CHARACTER COUNT
            MOVE    #2,D1      SET CHANNEL 2
            TRAP    1          PRINT THE TIME AND DATE AND CR
            DC.W    SVCOUL
            ....
```

This program segment first prints on channel 2 (usually the display) using SVCOUM (2) "THE DATE AND TIME IS: ", then reads the 9 character date using SVCDAT (30) followed by the 8 character time using SVCTIM into a buffer at BUFF, and then prints them followed by a carriage return on channel 2 from the buffer using SVCOUL (6).

NOTES: 1. The format of the time returned by SVCTIM is HH:MM:SS on a 24-hour clock.

2. If the MultI/O board is not installed or does not have the clock/calendar feature, the returned time will be 88:88:88.

3.2.25 SVCTOM 32 = $20

PURPOSE: To obtain the highest usable Datamover memory address.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: A0.L = Highest memory address that may be used

DESCRIPTION: SVCTOM is used to determine the highest usable memory address. User memory under DMXMON extends from $001000 through the value returned by SVCTOM. Typically, top-of-memory will be $03FFFF for a standard Datamover and CODOS 2.0. See the notes for exceptions.

```
EXAMPLE:    ....
            TRAP    1
            DC.W    SVCTOM     DETERMINE HIGHEST MEMORY ADDRESS
            CMPA    #20000,A0  TEST IF ENOUGH FOR IN-MEMORY SCRATCH
            BGE     INMEM      GO FOR IN-MEMORY SCRATCH IF SO
            ....               OPEN A SCRATCH FILE IF NOT
```

This program segment determines the top-of-memory and if it is less than 128K, sets up for using a scratch file on disk.

NOTES: 1. For DMXMON version 1.0, the top-of-memory value will always be $03FFFF unless it is patched to accomodate Datamovers with more or less than the standrad 256K of memory.

2. Versions of CODOS later than 2.0 are expected to make use of part of Datamover memory as a "virtual disk". The value returned by SVCTOM with the companion version of DMXMON will be dependent on the current size of the virtual disk.

The following SVCs all perform functions related to direct input and output of text. The SVCs call the MTU-130 I/O routines directly rather than through CODOS channel I/O as those in section 3.2 did and therefore offer greater flexibility at the expense of device dependence.

Many of these SVCs implicitly use the text cursor. The text cursor's location is defined by a pair of bytes that reside in 6502 memory. One of these bytes is the column (character) number the cursor is on and the other is the line number. Both of these start counting at one instead of zero. Since the cursor location bytes are not directly addressable by the 68000, SVCs are also provided to directly read and set the text cursor location.

ID #	MNEMONIC	FUNCTION
64	GETKEY	Wait until a new key is pressed and return its code
65	TSTKEY	Test if a new key is pressed, single recognition
66	IFKEY	Test if a key is pressed, multiple recognition
67	INLINE	Input a line from the keyboard with editing allowed
68	EDLINE	Display a line and allow editing using the keyboard
69	CLRDSP	Clear the entire 480 x 256 display screen
70	INITIO	Clear screen, initialize keyboard, restore I/O parameters
71	INITTW	Clear screen, restore I/O parameters
72	DEFTW	Specify the position and size of the text window
73	REDTW	Read the current size and position of the text window
74	CLRHTW	Clear the text window and home the cursor
75	CLRTW	Clear the text window only
76	CLRTLN	Clear a specified text line
77	CLREOL	Clear from text cursor to end of line
78	CLREOS	Clear from text cursor to end of text window
79	CLRLEG	Clear the legend boxes from the display
80	DRWLEG	Draw the function key legend boxes and the legends
81	OFFTCR	Turn the text cursor off
82	ONTCR	Turn the text cursor on
83	FLPTCR	Flip the video sense of the text cursor block
84	SETTCR	Arbitrarily set the text cursor position
85	READTCR	Read the current position of the text cursor
86	CRLF	Move the text cursor to left screen edge and down 1 line
87	HOMETW	Move the text cursor to upper left corner of text window
88	CURUP	Move the text cursor up 1 line if possible
89	CURLFT	Move the text cursor left 1 column if possible
90	CURRGT	Move the text cursor right 1 column if possible
91	CURDWN	Move the text cursor down 1 line & scroll if necessary
92	OUCH	Display a printable or interpret a control character
93	BEEP	Sound an arbitrary sounding beep

3.3.1 GETKEY 64 = $40

PURPOSE: Wait until a new key is struck and return its code.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.B = ASCII character code of key struck

DESCRIPTION: GETKEY will start scanning the MTU-130 keyboard and wait for the operator to press a key. During this time, a flashing cursor will appear on the display at the current text cursor location. The character typed is not echoed to the display. An audible click accompanies key depression if 6502 memory location $213 is $00 (it is $00 by default). See chapter 8 of the CODOS manual for additional parameters and details. The character code generated is MTU-130 extended ASCII (see Appendix H of the CODOS manual).

```
EXAMPLE:    ....
            TRAP    1         ; SVC 64 = GET A KEYSTROKE
            DC.W    64
            CMP.B   #$0D,D0   ; CHECK FOR CARRIAGE RETURN
            ....
```

3.3.2 TSTKEY 65 = $41

PURPOSE: Test if a new key is pressed; has multiple recognition lockout.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: CY 1 = a new key was pressed, 0 = no new key pressed
D0.B = ASCII character code of new key

DESCRIPTION: TSTKEY will scan the keyboard once looking for a key that is down. If one is found down that has not been previously recognized as down, its code is loaded into D0.B and the carry flag is set. If no new keys are found down, the carry flag is cleared and D0 is unchanged. The difference between this SVC and IFKEY is that a key is recognized as down only once until the operator releases it. This also applies to a key still down after recognition by GETKEY.

```
EXAMPLE:    ....
            TRAP    1         ; SVC 65 = TEST IF NEW KEY DOWN
            DC.W    65
            BCC     NOKEY     ; SKIP AHEAD IF NO KEY PRESSED
            MOVE.B  D0,KYCOD  ; PROCESS THE KEY CODE
            ....
```

3.3.3 IFKEY 66 = $42

PURPOSE: Test if a key is pressed without multiple recognition lockout.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: CY 1 = a key was pressed, 0 = no key pressed
D0.B = ASCII character code of key

DESCRIPTION: IFKEY will scan the keyboard once looking for a key that is down. If one is found down, its code is loaded into D0.B and the carry flag is set. If no keys are found down, the carry flag is cleared and D0 is unchanged. The difference between this SVC and TSTKEY is that a key may be repeatedly recognized as being down as long as the operator holds it down. There will be a click each time the key is recognized as being down unless 6502 memory location $0213 is set to $80.

```
EXAMPLE:        ....
                TRAP    1       ; SVC 66 = TEST IF KEY IS DOWN
                DC.W    66
                BCC     NOKEY   ; SKIP AHEAD IF NO KEY PRESSED
                MOVE.B  D0,KYCOD ; PROCESS THE KEY CODE
                ....
```

NOTES: 1. When several keys are simultaneously pressed, the one earliest in the keyboard scan sequence is recognized.

3.3.4 INLINE 67 = $43

PURPOSE: Input a line directly from the keyboard with editing and previous line recall supported.

ARGUMENTS PASSED: A0.L = Address of a buffer to receive the line entered

ARGUMENTS RETURNED: D0.B = Number of characters in the line less the CR
CY 1 = CTRL/Z entered, 0 = regular line entered

DESCRIPTION: INLINE accepts one line of text from the keyboard and allows all of the line-editing functions permitted by CODOS. Editing functions include CTRL/B for recalling prior lines, and automatic replacement of function keys with pre-defined macro character strings. Once started, INLINE will not return until a complete line has been entered, terminated by a carriage return or CTRL/Z (keyboard end-of-file) is entered. INLINE displays a cursor and echos characters typed to the screen.

```
EXAMPLE:        ....
                MOVEA.L #MYLBUF,A0 SET BUFFER ADDRESS
                TRAP    1          SVC 67 = INPUT AN ENTIRE LINE
                DC.W    67
                BCS     KEYEOF     JUMP IF CTRL/Z ENTERED
                TST.B   D0         TEST FOR NULL LINE
                ....
```

NOTES: 1. The difference between INLINE and SVCINL (5) is that INLINE accepts data only from the keyboard and it can use an arbitrary line buffer instead of the DMXMON system line buffer.

2. See EDLINE (68) for a list of editing characters recognized.

3. The maximun line length that may be input is 192 characters.

3.3.5 EDLINE 68 = $44

PURPOSE: To edit an entire line using the keyboard.

ARGUMENTS PASSED: A0.L = Address of a buffer containing the line to be edited
D0.B = Initial number of characters in the line less the CR

ARGUMENTS RETURNED: D0.B = Final number of characters in the line less the CR
CY 1 = CTRL/Z entered

DESCRIPTION: EDLINE is similar to INLINE (67) except that the input line buffer is assumed to already hold a line to be edited when EDLINE is called. The keyboard can be used to edit or accept the line. The full range of editing keys accepted by CODOS is available. To use EDLINE, load the address of the line into A0 and set D0 to the number of significant characters in the line. A CR character at the end of the line is not required. EDLINE will alter D0 to reflect the number of characters in the edited line.

EXAMPLE:

```
....
MOVEA.L #LINATT,A2  GET ADDRESS OF LINE ATTRIBUTE TABLE IN A2
MOVE.W  LINNO,D3    GET LINE NUMBER INTO D3
MULU    #5,D3       MULTIPLY LINE NUMBER BY 5
MOVEA.L 0(A2,D3),A0 SET BUFFER ADDRESS FROM TABLE INDEXED BY LINE#
MOVE.B  4(A2,D3),D0 SET CHARACTER COUNT FROM TABLE
TRAP    1           SVC 68 = EDIT THE LINE
DC.W    68
BCS     KEYEOF      JUMP IF CTRL/Z ENTERED
CMP.B   #0,D0       CHECK IF LINE DELETED
....
```

This program segment illustrates a simple line editing application where line number LINNO is presented to EDLINE for editing. The address and length of the line are taken from a line attribute table that the overall editing program maintains.

NOTES: 1. The use of EDLINE in conjunction with INLINE allows any interactive 68000 program to offer sophisticated command line and statement line editing with very little programming effort.

2. The editing characters accepted by EDLINE and INLINE are as follows:

BACK SPACE	Backspace 1 character
	Backspace 1 character
RUBOUT	Backspace 1 character then erase character under cursor
	Forward 1 space without erasing
space	Erase character under cursor then forward 1 space
INSERT	Enter insert mode, following characters are inserted
DELETE	Delete character under cursor and pull from right
SHIFT	Jump cursor to last character on line
SHIFT	Jump cursor to first character on line
CTRL/B	Delete current line and recall previous line(s)
CTRL/E	Toggle keyboard echo to display on or off
CTRL/R	Recall current line from buffer to display
CTRL/W	Delete from cursor to end of line inclusive
CTRL/X	Delete line and put cursor at beginning
CTRL/Z	End-of-file for keyboard input (first character only)

3. The maximum line length that may be edited is 192 characters.

3.3.6 CLRDSP 69 = $45

PURPOSE: To clear the entire 256 x 480 display screen.

ARGUMENTS: None

DESCRIPTION: CLRDSP will unconditionally clear the entire MTU-130 display screen to black including the legend display area at the screen bottom. If you want to clear only the text window, use CLRHTW (74) or OUTCHR with an argument of $0C (form-feed).

EXAMPLE:
```
....
TRAP    1           SVC 69 = CLEAR ENTIRE SCREEN
DC.W    69
....
```

This program segment will clear the entire screen.

3.3.7 INITIO 70 = $46

PURPOSE: To clear the screen, function key legends, and backup buffer and restore default I/O routine parameters.

ARGUMENTS: None

DESCRIPTION: INITIO will completely re-initialize the text display driver and is useful if its previous state is unknown. It performs the following functions:

1. Clears entire 480 x 256 screen area.
2. Clears all function key legends to blanks.
3. Clears function key substitution strings to nulls ($80).
4. Clears the "backup" buffer used to recall previous lines.
5. Draws the function key legend boxes.
6. Sets text window to 24 lines starting at the top of the screen.
7. The text cursor is placed in the home position (COL=1, LINE=1).
8. All text driver flags are set to their normal state.
9. The keyboard driver is initialized.
10. The audio DAC is initialized (may cause a speaker "pop").

EXAMPLE:
```
....
TRAP    1           SVC 70 = INITIALIZE TEXT I/O DRIVERS
DC.W    70
....
```

This program segment will initialize the text I/O drivers as described above.

3.3.8 INITTW 71 = $47

PURPOSE: To clear the screen and set default values of display parameters.

ARGUMENTS: None

DESCRIPTION: INITTW will completely re-initialize the text display driver and is useful it its previous state is unknown. It performs the following functions:

1. Clears entire 480 x 256 screen area.
2. Clears the "backup" buffer.
3. Draws the function key legend boxes and legends.
4. Sets text window to 24 lines starting at the top of the screen.
5. The text cursor is placed in the home position (COL=1, LINE=1).
6. All text driver flags are set to their normal state.
7. The audio DAC is initialized (may cause a speaker "pop").

```
EXAMPLE:    ....
            TRAP    1           SVC 71 = INITIALIZE TEXT WINDOW
            DC.W    71
            ....
```

This program segment will initialize the text I/O drivers as described above.

NOTES: 1. The differece between INITTW and INITIO is that INITTW does not affect the function key legends or substitution strings and it does not initialize the keyboard driver.

3.3.9 DEFTW 72 = $48

PURPOSE: To specify the position and size of the text window.

ARGUMENTS PASSED: D0.B = Number of text lines (1-25)
D1.W = 255-Y where Y is the y coordinate of the topmost text line.

ARGUMENTS RETURNED: None

DESCRIPTION: DEFTW is used to define the position and size of the text window. D0 gives the number of text lines in the window with an allowable range of 1 to 25 (24 if function key legends are to be displayed). D1 specifies the position of the top text line in terms of 255-Y where Y is the Y coordinate of the topmost dot of the top line characters. The width of the text window is fixed at 80 characters.

```
EXAMPLE:    ....
            MOVE.B  #16,D0      SPECIFY 16 TEXT LINES
            MOVE.W  #0,D1       STARTING AT TOP OF THE SCREEN
            TRAP    1           SVC 72 = SPECIFY SIZE AND LOC OF TEXT WINDOW
            DC.W    72
            ....
```

This program segment will set the text window to 16 lines starting at the top of the screen.

NOTES: 1. DEFTW will not move the text cursor itself but the next operation that does anything with the cursor will force it inside the new wondow if it was outside.

2. The requested number of text lines must fit between the specified top position and the bottom of the screen (Y=0) at the rate of 10 coordinate units per line.

3.3.10 REDTW 73 = $49

PURPOSE: To read the current position and size of the text window.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.B = Number of text lines (1-25)
D1.W = 255-Y where Y is the y coordinate of the topmost text line.

DESCRIPTION: REDTW is the complement of DEFTW and returns the current size and position of the text window. It is useful for saving the display state when a program is started so it can be restored on exit.

```
EXAMPLE:    ....
            TRAP    1           SVC 73 = READ SIZE AND LOC OF TEXT WINDOW
            DC.W    73
            MOVE.B  D0,OLDNL    SAVE NUMBER OF LINES
            MOVE.W  D1,OLDTL    SAVE POSITION OF TOP LINE
            ....
```

This program segment will save the current size and location of the text window so it can be restored later.

3.3.11 CLRHTW 74 = $4A

PURPOSE: To clear the text window and home the cursor.

ARGUMENTS: None

DESCRIPTION: CLRHTW will clear the currently defined text window move the cursor to the home position.

```
EXAMPLE:    ....
            TRAP    1           SVC 74 = CLEAR TEXT WINDOW AND HOME CURSOR
            DC.W    74
            ....
```

This program segment will clear the text window and home the text cursor. Display area outside the text window is not affected.

3.3.12 CLRTW 75 = $4B

PURPOSE: To clear the text window only.

ARGUMENTS: None

DESCRIPTION: CLRTW sill clear the currently defined text window. The cursor remains in its present position.

```
EXAMPLE:    ....
            TRAP    1           SVC 75 = CLEAR TEXT WINDOW AND HOME CURSOR
            DC.W    75
            ....
```

This program segment will clear the text window without affecting anything else.

3.3.13 CLRTLN 76 = $4C

PURPOSE: To clear a specified text line.

ARGUMENTS PASSED: D0.B = Line number to clear

ARGUMENTS RETURNED: None

DESCRIPTION: CLRTLN will clear the line specified by D0. The line specified must be inside the text window. Lines are numbered from the top line down starting with one. The text cursor position is not affected.

EXAMPLE:
```
....
MOVE.B  #1,D0       SET UP TO CLEAR THE TOP TEXT LINE
TRAP    1           SVC 76 = CLEAR TEXT LINE
DC.W    76
....
```

This program segment will clear the topmost text line in the text window.

3.3.14 CLREOL 77 = $4D

PURPOSE: To clear from the text cursor location to the end of the line the cursor is on.

ARGUMENTS: None

DESCRIPTION: CLREOL will clear the remaining portion of the text line the cursor is on. The character under the cursor is also cleared. This is useful when overwriting a text line with a new line that may be shorter than the old line.

EXAMPLE:
```
....
TRAP    1           SVC 77 = CLEAR REST OF LINE TO RIGHT OF CURSOR
DC.W    77
....
```

This program segment will clear all text to the right of the cursor.

3.3.15 CLREOS 78 = $4E

PURPOSE: To clear from the text cursor location to the bottom of the text window

ARGUMENTS: None

DESCRIPTION: CLREOS will clear the remaining portion of the text line the cursor is on. The character under the cursor is also cleared. Following this, all lines below the cursor to the bottom of the text window are cleared.

EXAMPLE:
```
....
TRAP    1           SVC 78 = CLEAR REST OF SCREEN BEYOND CURSOR
DC.W    78
....
```

This program segment will clear all text to the right and below the cursor.

3.3.16 CLRLEG 79 = $4F

PURPOSE: To clear the legend boxes from the display.

ARGUMENTS: None

DESCRIPTION: CLRLEG will clear the bottommost 16 scan lines of the screen which is normally used to display the function key legends.

```
EXAMPLE:        ....
                TRAP    1           SVC 79 = CLEAR LEGEND DISPLAY AREA
                DC.W    79
                ....
```

This program segment will clear the legend display area.

3.3.17 DRWLEG 80 = $50

PURPOSE: To draw the function key boxes and legends.

ARGUMENTS PASSED: A0.L = Address of 64 character string containing the legend text.

ARGUMENTS RETURNED: None

DESCRIPTION: DRWLEG erases the existing legend area (bottom 16 scan lines of the display area) and draws 8 legend boxes, each containing an 8 character function key legend. The boxes and legends are displayed in two groups of 4 like the function keys on the MTU-130 keyboard. The legends to be used are ASCII strings of exactly 8 characters each which are taken from a 64 byte buffer whose address is passed in A0. Non-displayable characters (control or bit 7 set) are treated as if they were blanks. The position of the text cursor is not affected.

```
EXAMPLE:        ....
                MOVEA.L #LEG3A,A0   SET ADDRESS OF LEGENDS FOR "OTHER" MENU
                TRAP    1           SVC 80 = DRAW THE LEGENDS
                DC.W    80
                ....
        LEG3A   DC.B    ' RECALL   FIND   MACRO   FIGURE '
                DC.B    'NEWPAGE  UPDATE   QUIT   OTHER  '
```

This program segment will change the function key legends to those in LEG3A.

3.3.18 OFFTCR 81 = $51

PURPOSE: To turn the text cursor off.

ARGUMENTS: None

DESCRIPTION: OFFTCR is used for direct program control of the text cursor. When controlling the text cursor directly, it should be turned off before it is moved. Direct program control of the cursor is necessary when using the interrupting keyboard feature described in section 4.5.

```
EXAMPLE:    ....
            TRAP    1           SVC 81 = TURN THE TEXT CURSOR OFF
            DC.W    81
            ....
```

This program segment will turn the text cursor off.

NOTES: 1. If the text cursor is already off, OFFTCR will have no effect.

3.3.19 ONTCR 82 = $52

PURPOSE: To turn the text cursor on.

ARGUMENTS: None

DESCRIPTION: ONTCR is used for direct program control of the text cursor. The text cursor is normally a reverse-video block that covers the character under the cursor. Turning the cursor on thus flips the video sense of the cursor block. To make the cursor flash, it must be alternately turned on and off or periodically flipped. Direct program control of the cursor is described in section 4.5.

```
EXAMPLE:    ....
            TRAP    1           SVC 82 = TURN THE TEXT CURSOR ON
            DC.W    82
            ....
```

This program segment will turn the text cursor on.

NOTES: 1. If the text cursor is already on, ONTCR will have no effect.

3.3.20 FLPTCR 83 = $53

PURPOSE: To flip the video sense of the text cursor block.

ARGUMENTS: None

DESCRIPTION: FLPTCR is used to flash the text cursor when performing direct program control of the text cursor. To make the cursor flash, it must be periodically flipped 2 to 6 times per second. Direct program control of the cursor is described in section 4.5.

```
EXAMPLE:    ....
            TRAP    1           TURN THE TEXT CURSOR ON
            DC.W    ONTCR
      FLIP  TRAP    1           TEST IF NEW KEYSTROKE
            DC.W    TSTKEY
            BCS     NWKY        JUMP OUT IF SO
            MOVE.W  #7,D0       WAIT 116MS = 4.3HZ FLASH RATE
            TRAP    1
            DC.W    WAIT
            TRAP    1           FLIP THE CURSOR
            DC.W    83
            JMP     FLIP        GO FOR ANOTHER CYCLE
      NWKY  TRAP    1           BE SURE CURSOR IS OFF BEFORE MESSING
            DC.W    OFFTCR
            ....
```

This program segment will turn the text cursor on and flash it at a 4.3Hz rate while testing the keyboard for a keystroke. When a keystroke is detected, the cursor is extinguished and the character processed.

NOTES: 1. If the text cursor is already on, ONTCR will have no effect.

3.3.21 SETTCR 84 = $54

PURPOSE: To arbitrarily set the text cursor position.

ARGUMENTS PASSED: D0.B = Column number (starts at 1)
D1.B = Line number (starts at 1)

DESCRIPTION: SETTCR is used to move the text cursor to an arbitrary location in the text window. Note that the line and column numbers start at 1, not zero.

EXAMPLE:

```
....
MOVE.B  #80,D0      SET COLUMN 80
MOVE.B  #3,D1       SET LINE 3
TRAP    1           MOVE THE CURSOR TO THE RIGHT EDGE OF LINE 3
DC.W    84
....
```

This program segment will move the text cursor to the rightmost character position of text line 3.

NOTES: 1. The new cursor position must be inside the text window. If it is not, it will be forced inside before it is used.

2. Both the line and column numbers start at 1, not zero.

3.3.22 READTCR 85 = $55

PURPOSE: To read the current position of the text cursor.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.B = Column number (leftmost is 1)
D1.B = Line number (topmost is 1)

DESCRIPTION: READTCR is used to determine the present text cursor position. It is useful for saving the cursor location while an error message is displayed, etc.

EXAMPLE:

```
....
TRAP    1           READ THE CURSOR POSITION
DC.W    85
MOVE.B  D0,CURLOC   SAVE THE CURSOR LOCATION
MOVE.B  D1,CURLOC+1
....
```

This program segment will read the current cursor location and save it as a pair of bytes at CURLOC.

3.3.23 CRLF 86 = $56

PURPOSE: To move the cursor to the left screen edge and down 1 line.

ARGUMENTS: None

DESCRIPTION: CRLF will move the text cursor to the left screen edge and down one text line. If it is already on the bottom line of the text window, the display will be scrolled upward one line instead. CRLF performs the same function as displaying a carriage return code ($0D) with OUCH (92).

EXAMPLE:
```
....
TRAP    1           DO A CR-LF
DC.W    86
....
```

This program segment will move the cursor down 1 line and to the left screen edge.

3.3.24 HOMETW 87 = $57

PURPOSE: To move the cursor to the upper left corner of the text window.

ARGUMENTS: None

DESCRIPTION: HOMETW will move the text cursor to the upper left corner of the text window. HOMETW performs the same function as displaying a $A4 code with OUCH ($92).

EXAMPLE:
```
....
TRAP    1           HOME THE TEXT CURSOR
DC.W    87
....
```

This program segment will move the cursor to the home position.

3.3.25 CURUP 88 = $58

PURPOSE: To move the cursor up 1 line if possible.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: CY set if move was not successful, clear if it was

DESCRIPTION: CURUP will move the text cursor up 1 text line if possible. The carry flag will be clear if the move was successful or set if the cursor is already on the top line. The display is not affected in either case.

EXAMPLE:
```
....
TRAP    1           MOVE THE TEXT CURSOR UP
DC.W    88
BCC     DIDMOV      JUMP IF THE MOVE WAS SUCCESSFUL
```

This program segment will try to move the cursor up one text line and then jump to DIDMOV if the move was successful.

3.3.26 CURLFT 89 = $59

PURPOSE: To move the cursor left 1 column if possible.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: CY set if move was not successful, clear if it was

DESCRIPTION: CURLFT will move the text cursor left 1 column if possible. The carry flag will be clear if the move was successful or set if the cursor is already at the left margin. The display is not affected in either case.

EXAMPLE:

```
....
TRAP    1           MOVE THE TEXT CURSOR LEFT
DC.W    89
BCC     DIDMOV      JUMP IF THE MOVE WAS SUCCESSFUL
```

This program segment will try to move the cursor left one column and then jump to DIDMOV if the move was successful.

3.3.27 CURRGT 90 = $5A

PURPOSE: To move the cursor right 1 column if possible.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: CY set if move was not successful, clear if it was

DESCRIPTION: CURRGT will move the text cursor right 1 column if possible. The carry flag will be clear if the move was successful or set if the cursor is already at the right margin. The display is not affected in either case.

EXAMPLE:

```
....
TRAP    1           MOVE THE TEXT CURSOR RIGHT
DC.W    90
BCC     DIDMOV      JUMP IF THE MOVE WAS SUCCESSFUL
```

This program segment will try to move the cursor right one column and then jump to DIDMOV if the move was successful.

3.3.28 CURDWN 91 = $5B

PURPOSE: To move the cursor down 1 line and scroll the display if necessary.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: CY set if the move resulted in a display scroll

DESCRIPTION: CURDWN will move the text cursor down 1 text line and scroll if necessary. The carry flag will be set if a scroll was necessary (cursor on bottom line) or clear if no scroll took place.

```
EXAMPLE:        ....
                TRAP    1           MOVE THE TEXT CURSOR UP
                DC.W    88
                BCS     DIDSCR      JUMP IF THE MOVE RESULTED IN A SCROLL
```

This program segment will move the cursor down one text line and then jump to DIDSCR if the move resulted in an upward scroll of the text window.

3.3.29 OUCH 92 = $5C

PURPOSE: To display a printable character or interpret a control character.

ARGUMENTS PASSED: D0.B = Character to display or interpret

ARGUMENTS RETURNED: None

DESCRIPTION: OUCH is the primary text display SVC. Through it, printable characters are displayed and most of the control functions can be effected with control characters. A complete list of the characters accepted by OUCH and their functions may be found in Appendix H of the CODOS manual.

```
EXAMPLE:        ....
                MOVE.B  (A2)+,D0    GET NEXT CHARACTER TO OUTPUT
                TST.B   D0
                BEQ     EOM         JUMP IF END OF MESSAGE
                TRAP    1           ELSE DISPLAY OR INTERPRET IT
                DC.W    92
                JMP     OUTXT       GO FOR ANOTHER
        EOM     ....
```

This program segment will display characters from a buffer pointed to by A2 until a null character ($00) is found.

3.3.30 BEEP 93 = $5D

PURPOSE: To sound an arbitrary sounding beep.

ARGUMENTS PASSED: D0.L = Sound parameters as follwos:
Low byte = pitch, 1-255
Mid low byte = volume, 0-127
Mid high byte = duration, 0-127
High byte is ignored

ARGUMENTS RETURNED: None

DESCRIPTION: BEEP will sound a tone in the MTU-130's speaker with controllable pitch, volume, and duration. The waveform is square. The pitch value specifies the waveform period in units of 200uS. The volume ranges from 0 (silence) to 127 with 64 being normal volume for an attention-getting beep. Duration is specified as the number of complete waveform cycles generated, therefore the duration in seconds is dependent on the pitch. Typical value for a polite beep is $000C4005 while a good value for a strident honk is $00507F20.

EXAMPLE:

```
....
MOVE.L  #$507F20    SOUND A "YOU BLEW IT BUDDY" BEEP
TRAP    1
DC.W    93
....
```

This program segment will sound a loud, irritating honk.

NOTES: 1. The sound of long beeps will be affected somewhat if the timer, counter, or interrupting keyboard is enabled.

3.4 GRAPHIC DISPLAY SVCS

The following SVCs all perform functions related to the input and output of graphic information. These SVCs call the MTU-130 graphic I/O routines directly since there is no channel I/O support through CODOS for graphic information.

Most of these SVCs implicitly use the graphic cursor. The graphic cursor's location is defined by a pair of 16 bit words that reside in 6502 memory. One of these words is the X coordinate and the other is the Y coordinate. For the present MTU-130 display implementation, the X coordinate may range from 0 through 479 and the Y coordinate may range from 0 through 255. Note that although the Y coordinate value can be represented with a single byte, a full word should be used as with the X coordinate to minimize compatibility problems with future display upgrades. Since the cursor location bytes are not directly addressable by the 68000, SVCs are also provided to directly read and set the graphic cursor location.

All graphic SVCs verify that the coordinates are in range before an operation is performed. If one or both are out of range, the offending coordinate is forced to the nearest limit. Note that this is not the same as windowing because line angles may be affected by the forcing.

ID #	MNEMONIC	FUNCTION
128	SGMODE	Set the value of GMODE
129	RGMODE	Read the current state of GMODE
130	SDSPAT	Set the dashed line pattern
131	RDSPAT	Read the current state of the dashed line pattern
132	SMOVE	Move the graphic cursor to a specified point
133	SDRAW	Draw a solid white line from graphic cursor to a point
134	SVEC	Draw a line from cursor to a point according to GMODE
135	SDOT	Plot a dot at a specified point according to GMODE
136	SMOVER	Move the graphic cursor relative to itself
137	SDRAWR	Draw a solid line from cursor to a point relative to itself
138	SVECR	Draw from cursor to a point relative to itself using GMODE
139	SDOTR	Plot a dot at a point relative to the cursor using GMODE
140	SDRWCH	Draw a single character at the graphiccursor position
141	SISDOT	Determine state of pixel at specified location
142	SONGC	Turn on the graphic crosshair cursor
143	SOFFGC	Turn off the graphic crosshair cursor
144	RDGCR	Read the position of the graphic cursor
145	SGRIN	Interactivly input a coordinate using the crosshair cursor
146	SLTPEN	Activate the light pen and return coordinates if a hit
147	SINTLP	Activate the light pen
148	STSLP	Test light pen and return coordinates if a hit

3.4.1 SGMODE 128 = $80

PURPOSE: Set the value of GMODE.

ARGUMENTS PASSED: D0.B = New value of GMODE

ARGUMENTS RETURNED: None

DESCRIPTION: GMODE is a byte of flags kept in 6502 memory which defines the type of line drawn by a number of the drawing SVCs. The table below lists the valid values for GMODE and their effect:

$00 = Move
$40 = Erase
$60 = Erase dashed
$80 = Draw
$A0 = Draw dashed
$C0 = Flip
$E0 = Flip dashed

EXAMPLE:

```
....
MOVE.B  #$A0,D0  ; SET GMODE FOR A WHITE DASHED LINE
TRAP    1        ; SVC 128 = SET GMODE
DC.W    128
....
```

This program segment will set GMODE for a white dashed line when drawing with SVEC or SVECR. When using SDOT or SDOTR, the dots plotted will be white.

NOTES: 1. The setting of GMODE affects only those SVCs that say they are affected.

2. The setting of GMODE is not preserved by SDRAW (133), SDRAWR (137), SONGC (142), SOFFGC (143), or SGRIN (145).

3.4.2 RGMODE 129 = $81

PURPOSE: Read the current setting of GMODE

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.B = Current value of GMODE

DESCRIPTION: GMODE is a byte of flags kept in 6502 memory which defines the type of line drawn by a number of the drawing SVCs. See SGMODE for a table of the valid values for GMODE and their effect.

EXAMPLE:

```
....
TRAP    1        SVC 129 = READ GMODE
DC.W    129
CMP.B   #$80,D0  TEST IF GMODE SET FOR SOLID WHITE LINES
BEQ     ISWHITE  BRANCH IF SO
....
```

This program segment will read the value of GMODE and branch to ISWHITE if it is set for white solid lines.

3.4.3 SDSPAT 130 = $82

PURPOSE: Set the dashed line pattern.

ARGUMENTS PASSED: D0.W = New contents of dash pattern register

ARGUMENTS RETURNED: None

DESCRIPTION: For certain values of GMODE ($60, $A0, and $E0), dashed lines are drawn. A 16 bit word kept in 6502 memory determines the on-off pattern that makes up the dashes. As dashed lines are drawn, the 16 bit pattern is circularly shifted left one bit for each pixel plotted. If a one is end-around-shifted, then the pixel is set/erased/flipped according to GMODE, otherwise, nothing is done at that pixel location.

EXAMPLE:

```
....
MOVE.W  #$6F6F,D0  SET A PATTERN WITH ALTERNATE SHORT AND LONG
TRAP    1          DASHES
DC.W    130
....
```

This program segment initializes the 16 bit dash pattern to alternate short and long dashes.

NOTES: 1. The normal default dash pattern is $F0F0 which produces dashes 4 pixels long separated by spaces 4 pixels long.

2. If the dash pattern is set to zero and GMODE specifies a dashed line, nothing will be drawn.

3.4.4 RDSPAT 131 = $83

PURPOSE: Read the current state of the dash pattern.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.W = Current state of dash pattern

DESCRIPTION: The dash pattern is a 16 bit word kept in 6502 memory that controls the drawing of dashed lines. RDSPAT is used for for saving this value while unrelated lines are drawn and then restoring it to continue drawing without a pattern discontinuity.

EXAMPLE:

```
....
TRAP    1          SVC 131 = READ DASH PATTERN
DC.W    131
MOVE.W  D0,OLDPAT  SAVE FOR USE LATER
....
```

This program segment reads the dash pattern and saves it in OLDPAT for later use.

NOTES: 1. The dash pattern value read by RDSPAT will not necessarily be the same value that was set by a prior SDSPAT SVC if dashed line drawing has taken place between setting and reading. Viewed circularly however, the pattern of one bits will be the same.

3.4.5 SMOVE 132 = $84

PURPOSE: Move the graphic cursor to a specified point.

ARGUMENTS PASSED: D0.W = X coordinate of new location
D1.W = Y coordinate of new location

ARGUMENTS RETURNED: None

DESCRIPTION: SMOVE simply moves the graphic cursor to the point specified by the two arguments. The contents of the screen are not affected in any way.

```
EXAMPLE:    ....
            MOVE.W  #247,D0   MOVE THE GRAPHIC CURSOR TO X=247, Y=135
            MOVE.W  #135,D1
            TRAP    1         SVC 132 = MOVE GRAPHIC CURSOR
            DC.W    132
            ....
```

This program segment moves the graphic cursor to X=247 and Y=135.

3.4.6 SDRAW 133 = $85

PURPOSE: Draw a solid line from the graphic cursor to a specified point.

ARGUMENTS PASSED: D0.W = X coordinate of next endpoint
D1.W = Y coordinate of next endpoint

ARGUMENTS RETURNED: None

DESCRIPTION: SDRAW draws a soild white line from the current graphic cursor location to the point specified by the arguments. After the line is drawn, the graphic cursor is moved to the point specified by the arguments.

```
EXAMPLE:    ....
            MOVE.W  #389,D0   DRAW FROM CURSOR LOCATION TO X=389, Y=67
            MOVE.W  #67,D1    AND THEN MOVE CURSOR TO X=389, Y=67
            TRAP    1         SVC 133 = DRAW TO DESIGNATED POINT
            DC.W    133
            ....
```

This program segment draws a solid white line from the present cursor location to X=389 Y=67 and then moves the cursor to X=389 Y=67.

NOTES: 1. SDRAW will set GMODE to solid white but it will not affect the dash pattern.

3.4.7 SVEC 134 = $86

PURPOSE: Draw a line from the graphic cursor to a specified point according to GMODE.

ARGUMENTS PASSED: D0.W = X coordinate of next endpoint
D1.W = Y coordinate of next endpoint

ARGUMENTS RETURNED: None

DESCRIPTION: SVEC is the same as SDRAW except that the type of line drawn is influenced by GMODE and possibly the dash pattern. See SGMODE (128) for the effect of various GMODE values.

EXAMPLE:
```
....
MOVE.B  #$40,D0   SET GMODE FOR ERASE
TRAP    1
DC.W    SGMODE
MOVE.W  #389,D0   ERASE FROM CURSOR LOCATION TO X=389, Y=67
MOVE.W  #67,D1    AND THEN MOVE CURSOR TO X=389, Y=67
TRAP    1         SVC 134 = DRAW TO DESIGNATED POINT
DC.W    134       ACCORDING TO GMODE
....
```

This program segment will erases (draws black) from the present cursor location to X=389 Y=67 and then moves the cursor to X=389 Y=67.

NOTES: 1. The dash pattern is affected only if GMODE is $60, $A0, or $E0.

3.4.8 SDOT 135 = $87

PURPOSE: Plot a dot at a specified point according to GMODE.

ARGUMENTS PASSED: D0.W = X coordinate of dot
D1.W = Y coordinate of dot

ARGUMENTS RETURNED: None

DESCRIPTION: SDOT will plot a single dot at the specified point according to the value of GMODE. After the dot is plotted, the cursor is moved to the specified point. The appearance of the dot depends on the setting of GMODE according to the following list:

$00 = Move (no display effect)
$40 = Erase (plot black dot)
$80 = Draw (plot white dot)
$C0 = Flip (change dot from black to white or white to black)

EXAMPLE:
```
....
MOVE.B  #$C0,D0   SET GMODE FOR FLIP
TRAP    1
DC.W    SGMODE
MOVE.W  #132,D0   FLIP DOT AT X=132, Y=240
MOVE.W  #240,D1
TRAP    1         SVC 135 = PLOT A DOT
DC.W    135
....
```

This program segment flips the black/white status of the dot at X=132, Y=240 and then moves the cursor to X=132 Y=240.

NOTES: 1. The dash pattern is not affected.

2. GMODE values with bit 5 set ($60, $A0, and $E0) perform just as if bit 5 was not set.

3.4.9 SMOVER 136 = $88

PURPOSE: Move the graphic cursor to a point relative to the graphic cursor.

ARGUMENTS PASSED: D0.W = X displacement to new location
D1.W = Y displacement to new location

ARGUMENTS RETURNED: None

DESCRIPTION: SMOVER is similar to SMOVE except that the argument values are added to the present cursor position to determine its new position. The arguments therefore are 16 bit signed values.

```
EXAMPLE:        ....
                MOVE.W  #30,D0      MOVE THE GRAPHIC CURSOR RIGHT 30 AND
                MOVE.W  #-20,D1     DOWN 20
                TRAP    1           SVC 136 = MOVE GRAPHIC CURSOR RELATIVE
                DC.W    136
                ....
```

This program segment will move the graphic cursor 30 units to the right and 20 units below its present position.

3.4.10 SDRAWR 137 = $89

PURPOSE: Draw a solid line from the graphic cursor to a point relative to the graphic cursor.

ARGUMENTS PASSED: D0.W = X displacement to next endpoint
D1.W = Y displacement to next endpoint

ARGUMENTS RETURNED: None

DESCRIPTION: SDRAWR is similar to SDRAW except that the argument values are added to the present cursor position to determine the line's next endpoint. The arguments therefore are 16 bit signed values. After drawing, the cursor is updated to the new endpoint.

```
EXAMPLE:        ....
                MOVE.W  #-5,D0      DRAW A DIAGONAL LINE 10 UNITS DOWN AND 5 UNITS
                MOVE.W  #-10,D1     TO THE LEFT OF THE GRAPHIC CURSOR
                TRAP    1           SVC 137 = DRAW RELATIVE
                DC.W    137
                ....
```

This program segment will draw a diagonal line extending 5 units to the left and 10 units below the present cursor position. After drawing, the cursor is moved 5 units to the left and 10 units below its original position.

NOTES: 1. SDRAWR will set GMODE for solid white but will have no effect on the dash pattern.

3.4.11 SVECR 138 = $8A

PURPOSE: Draw a line from the graphic cursor to a point relative to the graphic cursor according to GMODE.

ARGUMENTS PASSED: D0.W = X displacement to next endpoint
D1.W = Y displacement to next endpoint

ARGUMENTS RETURNED: None

DESCRIPTION: SVECR is similar to SVEC except that the argument values are added to the present cursor position to determine the line's next endpoint. The arguments therefore are 16 bit signed values. After drawing, the cursor is updated to the new endpoint.

EXAMPLE:

```
....
MOVE.W  #300,D0   DRAW A DIAGONAL LINE 150 UNITS DOWN AND
MOVE.W  #-150,D1  300 UNITS TO THE RIGHT OF THE GRAPHIC CURSOR
TRAP    1         SVC 138 = DRAW RELATIVE ACCORDING TO GMODE
DC.W    138
....
```

This program segment will draw a diagonal line extending 300 units to the right and 150 units below the present cursor position. The type of line will be determined by the present setting of GMODE.

NOTES: 1. The dash pattern is affected only if GMODE is $60, $A0, or $E0.

3.4.12 SDOTR 139 = $8B

PURPOSE: Plot a dot at a point relative to the graphic cursor according to GMODE.

ARGUMENTS PASSED: D0.W = X displacement to the dot
D1.W = Y displacement to the dot

ARGUMENTS RETURNED: None

DESCRIPTION: SDOTR is similar to SDOT except that the argument values are added to the present cursor position to determine the dot's location. The arguments therefore are 16 bit signed values. After plotting, the cursor is updated to the dot's position.

EXAMPLE:

```
....
MOVE.W  #1,D0     PLOT A DOT JUST ABOVE THE THE DOT LAST PLOTTED
CLR.W   D1        GMODE ASSUMED = $80 = PLOT WHITE DOT
TRAP    1         SVC 139 = PLOT DOT RELATIVE
DC.W    139
....
```

This program segment will plot a dot just above the dot last plotted. GMODE is assumed to be set for plotting white dots. After the dot is plotted, the cursor is moved up 1 pixel position.

3.4.13 SDRWCH 140 = $8C

PURPOSE: To draw a single character at the graphic cursor location.

ARGUMENTS PASSED: D2.B = ASCII character to be drawn

ARGUMENTS RETURNED: None

DESCRIPTION: SDRWCH may be used to draw characters at any arbitrary location on the screen, The character cell used is 6 dots wide by 10 dots high into which a character 5 dots wide by 7 dots high is written as illustrated below:

```
                  . . . . . . . . . . . . . . . . . . . . .
                  . . 0 . . . 0 0 0 0 . . . . . . . . 0 . . . . .
                  . 0 . 0 . . 0 . . . 0 . . . . . . . 0 . . . . .
                  0 . . . 0 . 0 . . . 0 . . 0 0 0 . . 0 . 0 0 . .
                  0 . . . 0 . 0 0 0 0 . . 0 . . . 0 . 0 0 . . 0 .
                  0 0 0 0 0 . 0 . . . 0 . 0 . . . 0 . 0 . . . 0 .
                  0 . . . 0 . 0 . . . 0 . 0 . . 0 0 . 0 . . . 0 .
                  0 . . . 0 . 0 0 0 0 . . . 0 0 . 0 . 0 . . . 0 .
   character      . . . . . . . . . . . . . . . . . 0 . . . . . . .
  coordinates---->. . . . . . . . . . . . . 0 0 0 . . . . . . . .
```

The character's coordinates refer to the lower left corner of the 6 by 10 cell. The character's baseline is normally 2 dot rows above the bottom of the cell but lower case characters with descenders (g,j,p,q,y) will extend down to the bottom of the cell. The entire 6 by 10 character cell is cleared before the character is drawn so it may be desirable to draw characters first and then any graphics that might overlap portions of the cell. After the character is drawn, the X coordinate of the cursor is incremented by 6 in preparation for another character. Thus labels for charts and graphs may be easily drawn by repeated use of SDRWCH. The cursor's X position must be between 0 and 474 inclusive and its Y position must be between 0 and 247 inclusive. If either is out of range, the character will not be drawn at all. Only printable ASCII character codes ($20-$7F) may be drawn, all other codes will not be drawn.

EXAMPLE:

```
....
MOVE.W  #125,D0   MOVE TO X=125, Y=240
MOVE.W  #240,D1
TRAP    1
DC.W    SMOVE
MOVE.B  #'Q',D2   DRAW THE LETTER "Q" THERE
TRAP    1
DC.W    140
....
```

This program segment will draw the letter "Q" with its lower left corner at X=125 and Y=240. After drawing, the cursor is moved 6 units right to X=131.

NOTES: 1. The cursor will not be returned and moved down when X reaches the right screen edge.

3.4.14 SISDOT 141 = $8D

PURPOSE: To determine whether the pixel at the specified location is on or off.

ARGUMENTS PASSED: D0.W = X coordinate of pixel to be tested
D1.W = Y coordinate fo pixel to be tested

ARGUMENTS RETURNED: CY = 1 if the pixel is set (white dot)

DESCRIPTION: SISDOT is used to test the state of a particular pixel. After the pixel is tested, the graphic cursor is move to the position just tested and the carry flag is set equal to the dot's status.

```
EXAMPLE:    ....
            MOVE.W  XLOC,D0   TEST THE PIXEL AT XLOC,YLOC
            MOVE.W  YLOC,D1
            TRAP    1         SVC 141 = TEST A PIXEL
            DC.W    141
            BCC     ISOFF     SKIP IF THE PIXEL IS OFF
            ....              PIXEL IS ON
```

This program segment looks at the pixel at XLOC,YLOC and branches to ISOFF if the pixel is off (black).

3.4.15 SONGC 142 = $8E

PURPOSE: To turn on the graphic crosshair cursor.

ARGUMENTS: None

DESCRIPTION: The graphic crosshair consists of a full screen height vertical line drawn at the X position of the cursor and a full screen width horizontal line drawn at the Y position. The crosshair is drawn in flip mode which means that parts of an image it covers will be restored when it is later turned off with the SOFFGC SVC. Flip mode also means that the cursor will show up regardless of the background color of the screen.

```
EXAMPLE:    ....
            MOVE.W  XLOC,D0   MOVE TO XLOC,YLOC
            MOVE.W  YLOC,D1
            TRAP    1
            DC.W    SMOVE
            TRAP    1
            DC.W    142       SVC 142 = TURN ON THE CROSSHAIR CURSOR
```

This program segment will display the crosshair cursor at XLOC,YLOC.

NOTES: 1. If the crosshair is already on, SONGC will have no effect.

2. SONGC will set GMODE for flip solid mode.

3.4.16 SOFFGC 143 = $8F

PURPOSE: To turn the graphic crosshair cursor off.

ARGUMENTS: None

DESCRIPTION: SOFFGC turns the graphic cursor off that had been previously turned on by SONGC. For proper operation, the location of the graphic cursor must be the same as it was when the crosshair was turned on.

```
EXAMPLE:    ....
            TRAP    1
            DC.W    143         SVC 143 = TURN THE CROSSHAIR CURSOR OFF
            ....
```

This program segment will cease displaying the crosshair cursor.

NOTES: 1. If the crosshair is already off, SONGC will have no effect.

2. SOFFGC will set GMODE for flip solid mode.

3.4.17 RDGCSR 144 = $90

PURPOSE: Read the position of the graphic cursor.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.W = X coordinate of the graphic cursor
D1.W = Y coordinate of the graphic cursor

DESCRIPTION: RDGCSR may be used to determine the present graphic cursor position so that it may be saved, used in computations, etc.

```
EXAMPLE:    ....
            TRAP    1           READ THE CURRENT CURSOR POSITION
            DC.W    144
            MULU    #3,D0       MULTIPLY THE X COORDINATE BY 3
            TRAP    1           MOVE TO THE NEW POSITION
            DC.W    SMOVE
            ....
```

This program segment will multiply the X coordinate of the graphic cursor by 3 and move there.

3.4.18 SGRIN 145 = $91

PURPOSE: To allow user coordinate input by maneuvering a cursor with the keyboard cursor control keys.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.W = X coordinate of final cursor position
D1.W = Y coordinate of final cursor position
D2.B = ASCII character entered by the operator

DESCRIPTION: SGRIN activates a rapidly blinking full-screen crosshair cursor which can be maneuvered using the cursor keys on the keyboard. It remains active until a non-cursor key is struck. It then returns the coordinates and ASCII code of the key struck. The shifted cursor control keys move the cursor 5 times as fast as the unshifted cursor keys. HOME is not considered to be a cursor key by SGRIN.

EXAMPLE:

```
....
TRAP    1         DO THE SGRIN FUNCTION
DC.W    145
TRAP    1         MOVE TO THE COORDINATES RETURNED
DC.W    SMOVE
CMP.B   #"S"      TEST IF "S" ENTERED
BEQ     ISS       JUMP IF SO
....
```

This program segment will display a crosshair cursor and allow the user to move it around the screen with the keyboard cursor keys. When a non-cursor key is pressed, the graphic cursor will be moved to the last position of the crosshair. If the operator used the "S" key to terminate the coordinate entry, the program will branch to ISS.

NOTES: 1. The crosshair is drawn in flip mode so it will show up regardless of the background.

2. The current graphic cursor position is not changed by SGRIN.

3. There may be interference between SGRIN and the interrupting keyboard if it is enabled.

3.4.19 SLTPEN 146 = $92

PURPOSE: Activate light pen for one frame and return coordinates of hit, if any.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.W = X coordinate of pen hit
D1.W = Y coordinate of pen hit
CY Set = the pen saw light and D0 and D1 are valid

DESCRIPTION: SLTPEN first waits for the end of the current screen scan and then begins checking for a light pen "hit" (response to light from the screen). If light is seen during the next screen scan, the X and Y coordinates of the beam position when the hit occured are placed in D0 and D1 and the carry flag is turned on. If no light is seen during the scan, the carry flag is turned off. The maximum amount of time before returning is 33 milliseconds when no light is detected. The time varies from less than 1 to a maximum of 32 milliseconds when light is detected. The X and Y coordinates returned have resolution to the pixel level but a random variation up to + or - 2 coordinate units can be expected.

```
EXAMPLE:        ....
          LPEN  TRAP    1           SVC 146 = SLTPEN
                DC.W    146
                BCC     LPEN        WAIT FOR LIGHTPEN HIT
                TRAP    1           MOVE TO THE COORDINATES RETURNED
                DC.W    SMOVE
                TRAP    1           FLIP THE DOT THERE (GMODE=$40)
                DC.W    SDOT
                ....
```

This program segment will wait indefinitely for a light pen hit. When the pen sees light, the graphic cursor will be moved to the light's location and the pixel there flipped.

NOTES: 1. The light pen will generally not respond to features less than 2 pixels wide horizontally. Thus single dots and vertical lines that are only one pixel wide will be invisible to the pen unless the screen brightness is very high.

2. Light pens will, of course, only respond to those areas of the screen that are lit up. One method of obtaining light pen "hits" from the entire display is to use a white background. Any drawing that is required could be done as black-on-white. Also, on some monitors, it may be possible to adjust the black, or background, level up high enough to get "hits" over the entire display.

3. D0 and D1 are unchanged if the carry is returned clear (no hit).

3.4.20 SINTLP 147 = $93

PURPOSE: Wait for end of frame and then activate the light pen.

ARGUMENTS: None

DESCRIPTION: SINTLP performs the first half of the SLTPEN function. It essentially resets the light pen and allows it to look for light while the user program is doing something else. STSLP can be used later to determine if the pen saw light in the intervening period.

```
EXAMPLE:        ....
                TRAP    1           ACTIVATE THE LIGHT PEN
                DC.W    147
                ....
```

This program segment will activate the light pen at the end of the current screen sweep.

NOTES: 1. The execution time of SINTLP ranges from near zero to 16.6 milliseconds.

2. Once the pen sees light after being activated by SINTLP, it will hold its coordinate information indefinitely until read by STSLP or is activated again by SINTLP or SLTPEN.

3.4.21 STSLP 148 = $94

PURPOSE: Test for light pen hit and return coordinates if so.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: D0.W = X coordinate of hit
D1.W = Y coordinate of hit
CY Set = hit registered and coordinates are valid

DESCRIPTION: STSLP performs the second half of the SLTPEN function. It makes an immediate test of the light pen "hit" status and then returns with the carry set and coordinates in D0 and D1 if a hit had occured since it was last activated. If no light had been seen, carry is returned clear and D0 and D1 are unchanged.

EXAMPLE:

```
....
TRAP    1          ACTIVATE THE LIGHT PEN
DC.W    SINTLP
....               LOTSAMESS
TRAP    1          TEST IF A LIGHT PEN HIT
DC.W    148
BCC     HITT       JUMP IF SO
....
```

This program segment activates the light pen, performs some other functions, and then tests if a hit occured in the intervening time. A jump to HITT is taken if it did.

NOTES: 1. Any amount of time may elapse between a light pen hit and when STSLP us used to test for the hit and compute the X and Y coordinates of the hit.

2. D0 and D1 are unchanged of the carry is returned clear.

The special function SVCs provide a number of unique functions apart from CODOS interface, direct text I/O, and graphics I/O.

ID #	MNEMONIC	FUNCTION
192	READM	Read from any 6502 addressable memory location
193	WRITEM	Write into any 6502 addressable memory location
194	BLKRD	Copy a block of 6502 memory into 68000 memory
195	BLKWRT	Copy a block of 68000 memory into 6502 memory
196	CALLM	Call an arbitrary 6502 subroutine
197	SETWD68	Set the 68000 virtual display window parameters
198	SETVP65	Set the 6502 display viewport parameters
199	RFSH1	Copy the virtual display window into the viewport once
200	RFSFSR	Start continuous background window to viewport copying
201	RFSHSP	Stop continuous background display copying
202	CPYDSP	Reverse copy the display viewport into the window
203	WAIT	Wait for a specified time period before continuing
204	CLKON	Start a periodic interrupt to the 68000
205	CLKOFF	Stop the periodic interrupt
206	CNTON	Start a periodic increment of long word at $000268
207	CNTOFF	Stop the periodic increment
208	KYBON	Start the automatically scanned interrupting keyboard
209	KYBOFF	Stop the interrupting keyboard

3.5.1 READM 192 = $C0

PURPOSE: To read the contents of any 6502 memory location.

ARGUMENTS PASSED: A0.L = 6502 memory address (18 bits)

ARGUMENTS RETURNED: D0.B = contents of that address

DESCRIPTION: READM allows the user's 68000 program to read the contents of any arbitrary 6502 memory or I/O register location. It is particularly useful for direct control of 6502 I/O devices and reading in-memory arguments returned by 6502 subroutines called via CALLM (see section 3.5.5).

EXAMPLE:

```
....
MOVEA.L  #$BFD0,A0   SET ADDRESS OF PORT B OF PARALLEL I/O
TRAP     1           READ THE PORT
DC.W     192
BTST     #3,D0       TEST BIT 3
....
```

This program segment reads location $BFD0 in the 6502's address space which is port B of the standard MTU-130 parallel interface.

NOTES: 1. A full 18 bit address may be specified where bits 16 and 17 encode the bank number from 0 to 3. Note that banks 2 and 3 will normally access Datamover memory from $000000 to $00FFFF (bank 2) and $010000 to $01FFFF (bank 3).

2. 6502 display memory can be read between $1C000 and $1FBFF.

3. READM requires approximately 411 microseconds to execute.

3.5.2 WRITEM 193 = $C1

PURPOSE: To write into any 6502 memory location.

ARGUMENTS PASSED: A0.L = 6502 memory address (18 bits)
D0.B = Data byte to be written

ARGUMENTS RETURNED: None

DESCRIPTION: WRITEM allows the user's 68000 program to write into any arbitrary 6502 memory or I/O register location. It is particularly useful for direct control of 6502 I/O devices and setting up in-memory arguments to 6502 subroutines called via CALLM (see section 3.5.5).

EXAMPLE:

```
....
MOVEA.L  #$BFDC,A0   SET ADDRESS OF PCR OF PARALLEL I/O
TRAP     1           READ THE PORT
DC.W     READM
BSET     #0,D0       SET BIT 0 = POSITIVE ACTIVE EDGE ON CA1
TRAP     1           WRITE BACK THE NEW PCR CONTENTS
DC.W     193
....
```

This program segment first reads location $BFDC in the 6502's address space which is the peripheral control register for the standard MTU-130 parallel interface. It then sets bit 0 in the data read and writes the updated value back into the same location.

NOTES: 1. A full 18 bit address may be specified where bits 16 and 17 encode the bank number from 0 to 3. Note that banks 2 and 3 will normally access Datamover memory from $000000 to $00FFFF (bank 2) and $010000 to $01FFFF (bank 3).

2. 6502 memory locations $E000-$FFFF are normally write protected since they hold the CODOS operating system. If any of these must be written into, you must first execute an UNPROTECT command using SVCMON (13).

3. WRITEM requires approximately 381 microseconds to execute.

3.5.3 BLKRD 194 = $C2

PURPOSE: To copy a block of 6502 memory into 68000 memory.

ARGUMENTS PASSED: A0.L = 6502 memory address (18 bits)
A1.L = 68000 memory address
D0.W = Number of bytes to copy

ARGUMENTS RETURNED: None

DESCRIPTION: BLKRD allows the user's 68000 program to quickly copy a block of 6502 memory into the 68000's address space. It is particularly useful for reading back arrays of data generated by 6502 subroutines called via CALLM (see section 3.5.5).

```
EXAMPLE:    ....
            MOVEA.L  #$A000,A0   SET 6502 MEMORY ADDRESS = $A000
            MOVEA.L  #ARRYA2,A1  SET 68000 MEMORY ADDRESS = ARRYA2
            MOVE.W   #2048,D0    SET BYTE COUNT
            TRAP     1           DO THE BLOCK READ
            DC.W     194
            ....
```

This program segment reads a block of 2048 bytes from 6502 memory starting at $A000 into 68000 memory starting at ARRYA2.

NOTES: 1. A full 18 bit 6502 address may be specified where bits 16 and 17 encode the bank number from 0 to 3. Note that banks 2 and 3 will normally access Datamover memory from $000000 to $00FFFF (bank 2) and $010000 to $01FFFF (bank 3).

2. The block source may not cross a bank boundary in the 6502's address space.

3. BLKRD is substantially faster than using READM in a loop. The copy speed is approximately 46uS per byte.

3.5.4 BLKWRT 195 = $C3

PURPOSE: To copy a block of 68000 memory into 6502 memory.

ARGUMENTS PASSED: A0.L = 6502 memory address (18 bits)
A1.L = 68000 memory address
D0.W = Number of bytes to copy

ARGUMENTS RETURNED: None

DESCRIPTION: BLKWRT allows the user's 68000 program to quickly copy a block of 68000 memory into the 6502's address space. It is particularly useful for filling arrays of data for 6502 subroutines called via CALLM (see section 3.5.5).

```
EXAMPLE:    ....
            MOVEA.L  #$9000,A0   SET 6502 MEMORY ADDRESS = $9000
            MOVEA.L  #ARRYA1,A1  SET 68000 MEMORY ADDRESS = ARRYA1
            MOVE.W   #2048,D0    SET BYTE COUNT
            TRAP     1           DO THE BLOCK WRITE
            DC.W     195
            ....
```

This program segment writes a block of 2048 bytes from 68000 memory starting at ARRYA1 into 6502 memory starting at $9000.

NOTES: 1. A full 18 bit 6502 address may be specified where bits 16 and 17 encode the bank number from 0 to 3. Note that banks 2 and 3 will normally access Datamover memory from $000000 to $00FFFF (bank 2) and $010000 to $01FFFF (bank 3).

2. The block destination may not cross a bank boundary in the 6502's address space.

3. BLKWRT is substantially faster than using WRITEM in a loop. The copy speed is approximately 46uS per byte.

3.5.5 CALLM 196 = $C4

PURPOSE: To call an arbitrary 6502 subroutine.

```
ARGUMENTS PASSED: A0.W = Subroutine address in 6502 memory
                  D0.B = Passed contents of 6502 A register
                  D1.B = Passed contents of 6502 X register
                  D2.B = Passed contents of 6502 Y register
                  CY   = Passed state of 6502 carry flag

ARGUMENTS RETURNED: D0.B = Returned contents of 6502 A register
                    D1.B = Returned contents of 6502 X register
                    D2.B = Returned contents of 6502 Y register
                    CY   = Returned state of 6502 carry flag
```

DESCRIPTION: CALLM allows the user's 68000 program to call a 6502 machine language subroutine with arguments passed and returned in the 6502 machine registers. Additional arguments may be passed in memory using READM and WRITEM SVCs.

```
EXAMPLE:        ....
                MOVEA.W  #OUTSMP,A0  PREPARE TO CALL OUTPUT SAMPLE SUBROUTINE
                MOVE.B   (A3)+,D0    PASS SAMPLE VALUE IN 6502 A REGISTER
                MOVE.B   #2,D1       PASS CHANNEL NUMBER IN 6502 X REGISTER
                TRAP     1           CALL THE SUBROUTINE
                DC.W     196
                BCS                  JUMP IF SUCCESSFUL SAMPLE OUTPUT
                ....                 OTHERWISE, ERROR
```

This program segment calls a user written 6502 subroutine at OUTSMP with arguments in the 6502's A and X registers. Upon return, it checks the carry flag which is set by the subroutine if its function was performed successfully.

NOTES: 1. The 6502 subroutine is assumed to reside in bank 0 therefore the subroutine address is only 16 bits.

2. The subroutine must make only balanced use of the 6502 stack and return with an RTS since the return path to DMXMON on the 6502 stack.

3. The 6502 subroutine may use CODOS SVCs if desired.

4. The call/return overhead of CALLM is approximately 427uS.

3.5.6 SETWD68 197 = $C5

PURPOSE: To set the 68000 display window parameters.

```
ARGUMENTS PASSED: A0.L = Address of lower left corner of the window
                  D0.B = Width of the window in bytes (1-60)
                  D1.W = Height of the window in pixels (1-256)
                  D2.W = Width of full image in bytes

ARGUMENTS RETURNED: None
```

DESCRIPTION:

SETWD68 in conjunction with SETVP65 (198) is used to specify parameters for the transfer of bit mapped graphic information from 68000 memory to the 6502's display memory. This allows a graphic image to be computed at high speed directly in 68000 memory and then copied to actual display memory in the 6502's address space for display.

Typically, the total image size in 68000 memory will be larger than the MTU-130's screen. Consequently, just a portion of the image will be displayed. This portion is called the window and its size and location in the overall image is what SETWD68 establishes. Note that the window may be specified to be smaller than the 480 pixels (60 bytes) width and 256 pixels height of the MTU-130 screen but may not be specified to be larger. See section 4.2 for a complete description of the virtual display.

EXAMPLE:

```
        ....
        MOVEA.L  #LLCORN,A0  SET TO DISPLAY LOWER LEFT CORNER OF IMAGE
        MOVE.B   #160/8,D0   SPECIFY WIDTH OF 160 PIXELS
        MOVE.W   #100,D1     SPECIFY HEIGHT 100 PIXELS
        MOVE.W   #1024/8,D2  IMAGE WIDTH IS 1024 PIXELS
        TRAP     1           SET THE WINDOW PARAMETERS
        DC.W     197
        ....
```

This program segment establishes a display window 160 pixels wide and 100 pixels high located at the lower left corner of a virtual image in 68000 memory that is 1024 pixels wide.

NOTES:

1. The copy window routine does not need to know the virtual image height.

2. The virtual image may be anywhere in 68000 memory and as large as desired but a 64K boundary must only occur between rows of image bytes.

3. Image organization in 68000 memory is the same as it is in 6502 memory, that is, ascending memory addresses go from left to right and top to bottom. The bits in a byte appear on the display with the most significant bit at the left.

4. The window specified should reside entirely inside the virtual image.

5. There is no check for image overflow beyond the bottom of the 6502 display screen. If the window is specified improperly, the character font table may be overwritten.

3.5.7 SETVP65 198 = $C6

PURPOSE: To set the 6502 display viewport parameters.

ARGUMENTS PASSED: D0.W = X coordinate of lower left corner of the viewport
D1.W = Y coordinate of lower left corner of the viewport

ARGUMENTS RETURNED: None

DESCRIPTION:

SETVP68 in conjunction with SETWD65 (197) is used to specify parameters for the transfer of bit mapped graphic information from 68000 memory to the 6502's display memory. This allows a graphic image to be computed at high speed directly in 68000 memory and then copied to actual display memory in the 6502's address space for display.

Typically, the total image size in 68000 memory will be larger than the MTU-130's screen. Consequently, just a portion of the image will be displayed. This portion is called the window and its size and location in the overall image is what SETWD68 establishes. The location of this window on the physical display screen is what SETVP65 establishes and is called the viewport. The size of the viewport is the same as the size of the window specified by SETWD68. The location of the viewport is specified by giving the X and Y coordinates of its lower left corner using standard MTU-130 display conventions. Note that if the X coordinate is not divisible by 8, the next lower value that is divisible by 8 is used. The viewport size and location should be such that no part of the viewport is outside the 480x256 display area of the screen. See section 4.2 for more information.

EXAMPLE:

```
        ....
        MOVE.W   #96,D0      SPECIFY LEFT EDGE OF VIEWPORT AT 96
        MOVE.W   #100,D1     SPECIFY BOTTOM EDGE OF VIEWPORT AT 100
        TRAP     1           SET THE VIEWPORT PARAMETERS
        DC.W     198
        ....
```

This program segment establishes a viewport whose lower left corner is at X=96 and Y=100. In conjunction with the example for SETWD68, the viewport will extend to X=96+160=256 for the right edge and Y=100+100=200 for the top edge.

NOTES: 1. Remember that the X coordinate should be divisable of 8. The viewport width will always be a multiple of 8. The Y coordinate and Y size may be any integer.

2. There is no check for image overflow beyond the top or the bottom of the 6502 display screen. If the window and viewport are specified improperly, the character font may be overwritten.

3.5.8 RFSH1 199 = $C7

PURPOSE: To copy the display window once into the display viewport.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: None

DESCRIPTION: The RFSH1 SVC will physically copy 68000 memory into the 6502's display memory. The source and destination addresses are determined from the window and viewport parameters previously established by SETWD68 (197) and SETVP65 (198). The copy routine used is highly efficient (and specialized) and will perform the copy many times faster than BLKWRT (195) would.

```
EXAMPLE:    ....
            (example for SETWD68)
            (example for SETVP65)
            TRAP    1           DO THE SCREEN COPY
            DC.W    199
            ....
```

This program segment establishes the window and viewport parameters and then copies the window from 68000 memory into the viewport in 6502 memory.

NOTES: 1. The window and viewport parameters need not be reset after the copy unless they are to change.

2. The copy time for the entire 480x256 display is approximately 250 milliseconds. Times are proportionally shorter for smaller windows. The location and length/width ratio of the window have only a small effect on copy time.

3.5.9 RFSHSR 200 = $C8

PURPOSE: To start continuous background copying of the display window into the viewport.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: None

DESCRIPTION: RFSHR allows the programmer to begin a "background task" in the 6502 which continuously copies a display window in 68000 memory to a viewport in 6502 display memory. This action is performed continuously at maximum speed when the 6502 would otherwise be "idle" waiting for an I/O command from the 68000. RFSHR makes a copy of the parameters set by previous SETWD68 and SETVP65 SVCs when it is executed. Thus, these parameters may be changed after the background refresh is started so that an RFSH1 SVC can be executed for a single "foreground update" of a different portion of the screen without interfering with the ongoing background update.

```
EXAMPLE:    ....
            (example for SETWD68)
            (example for SETVP65)
            TRAP    1           START AUTOMATIC SCREEN COPY
            DC.W    200
            ....
```

This program segment establishes the window and viewport parameters and then initiates automatic copying of the window to the viewport until stopped by RFSHSP.

NOTES: 1. The automatic copy temporarily ceases while an SVC is being performed by the 6502.

2. When automatic screen update is enabled, the 6502 checks for an SVC request after each scan line is copied. The latency time thus introduced is a maximum of 900uS for a window 480 wide (60 bytes) and decreases for narrower windows at the rate of 14uS per byte. Therefore, if a lot of short execution time SVCs (such as READM, WRITEM or byte-at-a-time I/O) are to be executed and speed is important, the automatic screen update should be turned off while they are executed.

3. If the program is interrupted by a breakpoint or the INT key, the display will not be refreshed while DMXMON is in console mode. Automatic refresh will however resume when DMXMON returns to execution mode unless a RESET command is entered.

4. At the completion of each automatic refresh cycle, 68000 memory location $000141 will be set to a non-zero value. If location $000151 is non-zero, the 68000 will also be interrupted at this time. See sections 4.2 and 4.4 for additional information.

3.5.9 RFSHSP 201 = $C9

PURPOSE: Stop continuous background copying of the display window into the viewport at the end of the current frame.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: None

DESCRIPTION: RFSHSP will stop automatic background display copying started by RFSHSR. It will wait until the copy currently in progress completes before stopping to prevent partiallly updated images on the screen.

EXAMPLE:

```
....
(example for SETWD68)
(example for SETVP65)
TRAP    1           START AUTOMATIC SCREEN COPY
DC.W    RFSHSR
....
TRAP    1           STOP AUTOMATIC SCREEN COPY AT END OF FRAME
DC.W    201
....
```

This program segment establishes the window and viewport parameters and then initiates automatic copying of the window to the viewport. Later, it stops the automatic copying at the end of a copy cycle.

NOTES: 1. You may issue RFSHSP even if automatic copy had already been stopped or had never been started.

3.5.11 CPYDSP 202 = $CA

PURPOSE: To reverse copy the display viewport into the display window.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: None

DESCRIPTION: The CPYDSP SVC will copy 6502 memory into 68000 memory. The source and destination addresses are determined from the window and viewport parameters previously established by SETWD68 (197) and SETVP65 (198). The copy routine used is highly efficient (and specialized) and will perform the copy many times faster than BLKRD (194) would.

```
EXAMPLE:    ....                GENERATE IMAGE IN 6502 MEMORY
            ....                ESTABLISH WINDOW AND VIEWPORT PARAMETERS
            TRAP    1           COPY DISPLAY INTO MY MEMORY FOR PROCESSING
            DC.W    202
            ....
```

This program segment generates an image in 6502 display memory (perhaps by reading in a graphic image from disk), establishes the window and viewport parameters and then copies the image from 6502 memory into the window in 68000 memory.

NOTES: 1. The window and viewport parameters need not be reset after the copy unless they are to change.

2. The copy time for the entire 480x256 display is approximately 250 milliseconds. Times are proportionally shorter for smaller windows. The location and length/width ratio of the window have only a small effect on copy time.

3.5.12 WAIT 203 = $CB

PURPOSE: To wait for a specified time before continuing.

ARGUMENTS PASSED: D0.W = Wait time in 60ths of a second ("Jiffies").

ARGUMENTS RETURNED: None

DESCRIPTION: WAIT allows a 68000 program to wait for a precise amount of time before continuing. Because of memory refresh and interference from the 6502 accessing Datamover memory, the execution time of a given 68000 routine cannot be predicted precisely and therefore WAIT is useful if accuracy is desired.

```
EXAMPLE:    MOVE.W  #10*60,D0   SET 10 SECOND WAIT
            TRAP    1           DO THE WAIT
            DC.W    203
            ....
```

This program segment waits for 10 seconds and continues. The accuracy of the wait is unaffected by other activity in the system.

NOTES: 1. The maximum wait that may be specified is 65535/60 seconds or about 18 minutes.

2. The timing reference used is the vertical retrace signal from the MTU-130 display. The actual period of this signal is 16.67800 milliseconds ±.01%. This represents an error of +.07% with respect to 1/60 second (16.66666 milliseconds). .01% is 8.6 seconds per day and .07% is one minute per day.

3.5.13 CLKON 204 = $CC

PURPOSE: To start a periodic interrupt to the 68000.

ARGUMENTS PASSED: D0.W = Time between interrupts in 1/60ths of a second

ARGUMENTS RETURNED: None

DESCRIPTION: CLKON will start a periodic interrupt which a 68000 program may use to facilitate multi-tasking programming. At the end of each timing period, 68000 memory location $000143 will be set non-zero. If location $000153 is non-zero, an actual 68000 interrupt will be generated. See section 4.1 for more information.

```
EXAMPLE:    MOVE.W   #6,D0        SET 10 PER SECOND INTERRUPT FREQUENCY
            TRAP     1            START THE INTERRUPTS
            DC.W     204
            ....
```

This program segment starts a periodic interrupt every 100 milliseconds (10 per second).

NOTES: 1. The maximum period that may be specified is 65535/60 seconds or about 18 minutes.

2. The timing reference used is the vertical retrace signal from the MTU-130 display. The actual period of this signal is 16.67800 milliseconds ±.01%. This represents an error of +.07% with respect to 1/60 second (16.66666 milliseconds). .01% is 8.6 seconds per day and .07% is one minute per day.

3. Timer interrupts will continue to occur even if the program is interrupted by a breakpoint or by pressing the INT key or it exits normally. The RESET command will stop the timer.

4. Leaving DMXMON with a BYE or CODOS command will stop the timer interrupts.

5. The periodic interrupt clock is independent of the periodic counter described in section 3.5.15.

6. Having the periodic interrupt enabled will affect the sound of keyboard clicks slightly.

3.5.13 CLKOFF 205 = $CD

PURPOSE: To stop the periodic interrupt.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: None

DESCRIPTION: CLKOFF will stop a periodic interrupt that had been started by CLKON. It is good practice to stop the clock before the program exits.

```
EXAMPLE:    TRAP    1           STOP PERIODIC INTERRUPTS
            DC.W    205
            ....
```

This program segment stops the periodic interrupt started earlier.

NOTES: 1. You may use CLKOFF even if the clock had already been stopped or had never been started.

3.5.15 CNTON 206 = $CE

PURPOSE: To start periodic increment of a long word at $000268.

ARGUMENTS PASSED: D0.W = Time between increments in 1/60ths of a second

ARGUMENTS RETURNED: None

DESCRIPTION: CLKON will start periodic incrementing of the long word at $000268. The time between increments may be specified to 1/60 second resolution. This is useful for determining the execution time of program segments and for implementing software time-of-day clocks. Timing will remain accurate even if DMXMON enters console mode unless a RESET, BYE, or CODOS command is executed.

```
EXAMPLE:    CLR.L   $268        CLEAR THE COUNTER
            MOVE.W  #1,D0       SET 60 PER SECOND INCREMENT FREQUENCY
            TRAP    1           START INCREMENTING THE COUNTER
            DC.W    206
            ....
     READC  MOVE.L  $268,D0     READ THE COUNTER
            MOVE.W  #120        WAIT 60 MICROSECONDS
     READW  DBRN    D1,READW
            CMP.L   $268,D0     VERIFY IT
            BNE     READC       TRY AGAIN IF NOT VERIFIED
            ....
```

This program segment clears the counter at $000268 to zero and then sets a 60Hz update rate. It later reads the counter and verifies its accuracy (see note 5).

NOTES: 1. The maximum update period that may be specified is 65535/60 seconds or about 18 minutes. Zero is interpreted as 65536/60.

2. The timing reference used is the vertical retrace signal from the MTU-130 display. The actual period of this signal is 16.67800 milliseconds ±.01%. This represents an error of +.07% with respect to 1/60 second (16.66666 milliseconds). .01% is 8.6 seconds per day and .07% is one minute per day.

3. The counter will continue to be incremented even if the program is interrupted by a breakpoint or by pressing the INT key or it exits normally. The RESET command will stop the timer.

4. Leaving DMXMON with a BYE or CODOS command will stop the counter update.

5. Update of the 4 counter bytes is performed sequentially by the 6502 therefore there is a small chance that reading the counter will "catch" it partially updated. This possible error may be prevented by reading the counter, waiting at least 60 microseconds with a timed loop, and reading the counter again. If the two values read are equal, then they are accurate. If not, repeat the procedure. (see the example above).

6. Updating the counter is independent of the periodic interrupt.

7. Having the counter enabled will affect the sound of keyboard clicks slightly.

8. It takes over 2 years for the clock word to "wrap around" at 60 updates per second.

3.5.15 CNTOFF 207 = $CF

PURPOSE: To stop the periodic counter update.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: None

DESCRIPTION: CNTOFF will stop a periodic counter increment that had been started by CNTON.

```
EXAMPLE:    TRAP    1           STOP PERIODIC COUNTER UPDATE
            DC.W    207
            ....
```

This program segment stops the periodic counter update started earlier.

NOTES: 1. You may use CNTOFF even if the counter had already been stopped or had never been started.

3.5.17 KYBON 208 = $D0

PURPOSE: To start the automatically scanned interrupting keyboard.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: None

DESCRIPTION: KYBON will start an automatic scan of the MTU-130 keyboard so keystrokes can be entered into a 68000 program even if computation or other I/O activity (such as disk access for an editor) is going on. While the automatic keyboard scan is enabled, every keystroke entered will be unconditionally stored in 68000 memory location $000253 and location $000142 will be set non-zero. If location $000152 is non-zero, an actual 68000 interrupt will then be generated. Typical usage would be with a 68000 interrupt service routine that reads the keystrokes and places them in a programmed queue for action by the 68000 main program. See section 4.4 for more information.

```
EXAMPLE:    CLR.B    $142        CLEAR KEYSTROKE ENTERED FLAG
            TRAP     1           START THE INTERRUPTING KEYBOARD
            DC.W     208
            ....
```

This program segment starts the interrupting keyboard.

NOTES: 1. The interrupting keyboard employs an N-key rollover scan algorithm instead of the 2-key method the standard "wait on keystroke" routines do. It also produces a uniform "click" sound from the keys even if the timer or counter has been enabled. These features are advantageous for high speed input of lots of text by experienced typists, therefore use of the interrupting keyboard is recommended for 68000 based editors and word processors.

2. In general, the interrupting keyboard will not drop keystrokes even if several keys are hit all at once. The minimum time between keystrokes passed to the 68000 is 16 milliseconds however. If necessary, the keyboard routine will automatically queue keystrokes that are recognized more rapidly than this.

3. The interrupting keyboard does NOT flash or manipulate the cursor on the display or echo characters to the screen. These functions must be done by the 68000 program through the text display SVCs (see section 3.3 and 4.5).

4. A regular keyboard input SVC can be issued even if the interrupting keyboard is enabled. Keystrokes will be returned both by the SVC and by the interrupting keyboard routine. This condition can be recognized because each key pressed will produce two clicks.

5. If the program is interrupted by a breakpoint or by pressing the INT key, the interrupting keyboard will continue to enter keystrokes into the 68000 and interrupt it. This condition can be recognized because each key pressed will produce two clicks, one from the standard keyboard routine and one from the interrupting routine. Although disconcerting to hear, each routine will still function normally. The RESET command will stop the interrupting keyboard.

3.5.17 KYBOFF 209 = $D1

PURPOSE: To stop the interrupting keyboard.

ARGUMENTS PASSED: None

ARGUMENTS RETURNED: None

DESCRIPTION: KYBOFF will stop the interrupting keyboard that had been started by KYBON. It is good practice to stop the interrupting keyboard before the program exits.

```
EXAMPLE:    TRAP    1           STOP INTERRUPTING KEYBOARD
            DC.W    209
            ....
```

This program segment stops the interrupting keyboard started earlier.

NOTES: 1. You may use KYBOFF even if the keyboard had already been stopped or had never been started.

4. ADVANCED PROGRAMMING TECHNIQUES

DMXMON has a number of advanced features that allow usage of sophisticated programming techniques by advanced programmers. They are however implemented in such a manner that inherently simple programming tasks are not complicated. These advanced features include timing functions, automatic updating of the MTU-130 display from image data in 68000 memory, overlapped computation and I/O, and the handling of interrupts. Each of these topics is discussed in detail in the following sections.

4.1 USING THE TIMING FUNCTIONS

In order to satisfy a variety of timing requirements in 68000 programs, 3 different timing functions are offered. Each operates independently so a given program could use one, two, or all three simultaneously without interference. All of the timing functions use the 60Hz signal from the MTU-130 display generator as the basic timing signal. Timing resolution is therefore 1/60 second and timing accuracy is unaffected by other activity in the system. The timing functions may be used for applications ranging from simple waits and time-of-day clocks, to complex scheduling in multi-tasking operating systems built on top of DMXMON.

4.1.1 The WAIT SVC

The simplest timing function is that offered by the WAIT SVC which is SVC #203. With this SVC you simply specify the wait time in units of 1/60 second and the SVC processor will wait that amount of time before returning control to the user's program. This is a "wait once" function because each time a wait is required, the duration must be specified and the WAIT SVC executed. Additional information on WAIT may be found in the writeup on WAIT in section 3.5.12.

4.1.2 The Autoincrement Counter

Another kind of timing function is offered by the automatically incremented counter. The counter consists of a long word at 68000 memory location $000268. When the autoincrement counter is enabled, this long word will be periodically incremented at the specified interval. An SVC is provided to start the counter (CNTON) and to stop the counter (CNTOFF). The count interval, in 1/60ths of a second, is specified when the counter is started by CNTON. The user program may read and set the counter at any time.

A typical use for the autoincrement counter is as a time-of-day clock (in systems without a MultI/O board). The 68000 application program would typically ask the user for the time and date and then store a 32 bit number representing that time into the counter at $000268. The counter would then be started with a CNTON SVC and the update rate would be specified as 1 second (60). The contents of the counter then would reflect the current time and date until the system was powered down or DMXMON was exited with a BYE or CODOS command. Note that a 32 bit counter incremented at a one second rate will run for 136 years before overflowing.

The user program may read or set the counter at any time with a simple MOVE or other instruction. However, since the counter is incremented one byte at a time by the 6502, there is a small chance that the 6502 will be incrementing the counter at the same time and thus cause an erroneous reading or setting. The simplest method of avoiding this possibility when reading the counter is to read it twice with a timed loop delay of 60uS between readings and compare the readings. If they are equal, the value is accurate. If they differ, it should be read again until two successive readings are identical. The easiest method of insuring accurate writing is to immediately preceed the write operation with a WAIT SVC that specifies a 1/60 second wait. Since the counter update will always immediately preceed the wait expiration, there will be no chance of an error.

4.1.3 Periodic Interrupt Clock

The third timing technique is a periodic interrupt clock. With this technique, the 68000 can be interrupted at a set interval to perform functions such as task swapping and real-time control. When the periodic interrupt clock is started with the CLKON SVC, an interval is specified and an internal clock is set for that interval. Control is immediately returned to the user program after the clock is started. When the internal clock expires, it is immediately restarted (for the next interval). Simultaneously, 68000 memory location $000143 is unconditionally set non-zero to indicate that the clock has timed out and been restarted. If location $000153 is non-zero, the 68000 will also be interrupted. Thus a program can operate the periodic interrupt clock on a polled basis (wait in a loop for $143 to go non-zero) or a vectored interrupt basis. The handling of interrupts by DMXMON is described in section 4.4. Typically, after the user program has recognized the clock timeout, it would clear location $000143 in preparation for the next timeout. Since the clock is automatically restarted, the interrupt frequency is not affected by the interrupt response time of the program.

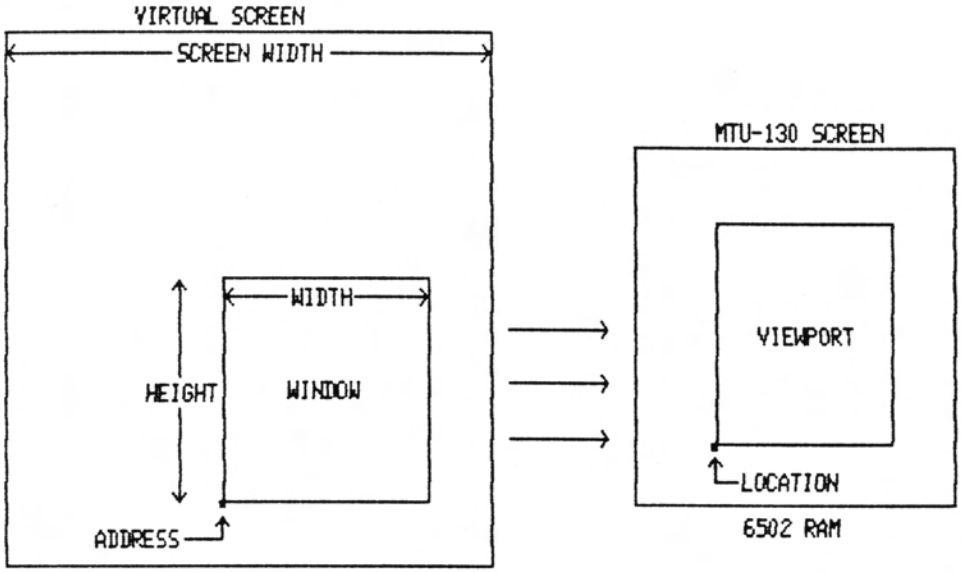
The periodic interrupt may be stopped by executing a CLKOFF SVC. This SVC stops the clock immediately so if only a single interrupt after a timeout is desired, you would just turn the clock off in the clock interrupt service routine. See section 4.4 for additional information on how DMXMON handles interrupts and links to user service routines.

4.2 USING THE 68000 MEMORY RESIDENT VIRTUAL DISPLAY

One of the most valuable features of the MTU-130's 6502 CPU board is the 15K block of memory set aside exclusively for a 480 wide by 256 high bit-mapped graphic display. Numerous SVCs in DMXMON support this display with a complete array of character and graphic drawing functions. All of these SVCs are actually executed by the 6502 microprocessor since it has direct access to the display memory.

However, DMXMON also has support for a virtual display memory that is directly addressable by the 68000. This allows the 68000's high potential drawing speed to be utilized in the generation of very complex graphic images. To see the contents of the virtual display memory, it is simply copied into the real display memory using an SVC, a process that takes less than a quarter second. The most significant feature of the virtual display memory however is that its size (width and height in pixels) can be specified by the programmer. In particular, the size can be much larger than the 480x256 size of the real display memory! In fact, the only real limitation is the memory capacity of the Datamover but even just half of the standard capacity (128K) can give a 1024x1024 virtual image size.

Of course if the virtual display memory is larger than the real display memory, only a portion of it may be viewed on the screen at once. In addition, the portion to be viewed may be smaller than the 480x256 screen in order to leave space for other uses, such as message display. SVCs are available to communicate the overall virtual display size, the location and size of that portion that is to be displayed, and where on the physical screen it is to be displayed. The drawing below will serve to illustrate the concepts and terminology that will be used in the following discussion.



The portion of 68000 memory that is used for the virtual display is called the virtual screen. The virtual screen's size is characterized solely by its width in bytes. The width in bytes is simply the width in pixels divided by 8. The virtual display SVCs can deduce the virtual display's height and location in 68000 memory from other information described below.

The portion of the virtual screen that will be visible on the real screen is called the window. The window is characterized by giving its width in bytes, its height in pixels and the 68000 memory address of its lower left corner. The window must not be specified to be larger than the MTU-130's real display which is 60 bytes wide and 256 pixels high. It is also the programmer's responsibility to ensure that the window stays inside the virtual screen; otherwise random data outside the virtual screen will show up on the real screen.

The portion of the real screen that will be used to display the window is called the viewport. The viewport is characterized simply by giving the X and Y coordinates of its lower left corner. The viewport's width and height is equal to the window's width and height. Note that the X coordinate given for the viewport location should be a multiple of 8. If it is not, it is rounded down to the next lower value that is a multiple of 8.

Before the window can be correctly copied into the viewport to be seen, all of the characteristics described above must be "Known" to the screen copy routines. Two SVCs are used to communicate this information. SETWD68 is the SVC used to establish information about the window. The arguments of SETWD68 are:

1. The width of the virtual screen in bytes
2. The width of the window in bytes
3. The height of the window
4. The 68000 memory address of the lower left corner of the window

SETVP65 is the SVC used to establish information about the viewport. The arguments of SETVP65 are:

1. The X coordinate of the lower left corner of the viewport
2. The Y coordinate of the lower left corner of the viewport

Once this information is established, the screen copy SVCs can be used as many times as desired until one of the parameters needs to be changed. Note that there is no error checking of the parameters specified. You need to be particularly careful to prevent the viewport from going beyond the MTU-130's screen boundaries. If it does, bank 1 memory below the screen and possibly a little above can be wiped out.

The simplest virtual display SVC is RFSH1. When this SVC is used, the window will be copied into the viewport once and the SVC will be finished. Nothing on the screen outside the viewport will be disturbed but everything inside will be replaced with the window contents. RFSH1 requires approximately 1/4 second for a full screen transfer (60 bytes wide and 256 pixels high) and proportionally less for smaller windows.

In cases where the virtual display is changing, one might want the real screen to be continuously updated from the virtual screen, preferably as an automatic background operation. This function is in fact available and can be started by executing the RFSHSR SVC. When RFSHSR is executed, a duplicate is made of all of the window and viewport parameters that were established by SETWD68 and SETVP65. After the duplicate is made, the RFSHSR SVC is considered complete. From then on the window is repeatedly copied into the viewport until an RFSHSP SVC is executed which stops the copying. Copying is actually done by the 6502 CPU and only steals about 5% of the available memory cycles from the 68000. Copying is temporarily suspended when another SVC is being executed but then restarts automatically when it is finished. When copying is stopped with RFSHSP, the SVC processor ensures that it will be stopped at the end of a complete copy cycle.

Since the automatic background copy routine has its own copy of the window and viewport parameters, these parameters may be changed after the automatic refresh has been started. Then RFSH1 SVCs can used to update a different viewport on the 130's screen on demand even while automatic refresh is taking place. This capability can be used to tremendous advantage in split-screen and multiple viewport applications.

There is also an SVC, CPYDSP, that does a reverse copy from the viewport into the window. It is useful where an image created by the 6502 (such as loading an image file from disk) needs to be transferred to the virtual screen for direct manipulation by the 68000. Note that all of these screen copying functions could be performed by the BLKRD and BLKWRT memory transfer SVCS but these are many times slower than the highly optimized routines just described. Also note that DMXMON does not provide support for drawing graphics and characters into the virtual screen in 68000 memory. Such routines must be provided by a separate program.

4.3 OVERLAPPING COMPUTATION AND I/O

Frequently the performance of a program can be substantially improved by overlapping computation and I/O operations. The technique is particularly effective when overall execution time is evenly split between waiting on I/O and computation and the I/O is relatively simple such as sequentially reading through a long file. In such cases, overlapping can nearly double the speed of a program.

The internal processing of SVCs by DMXMON actually consists of three phases. In the first phase, the arguments of the SVC are transferred to the 6502 by the 68000 portion of DMXMON and the I/O operation is started. In the second phase, 68000 DMXMON waits for an indication from the 6502 that the operation is complete. The third phase consists of transferring the return arguments back and returning to the user program that issued the SVC. Obviously, during the second phase, which can range from a few microseconds to several seconds depending on the operation, the 68000 is simply waiting when it could possibly be doing serious computing.

When a regular TRAP 1 SVC is executed, the start-wait-complete sequence just described is executed. When an SVC using a TRAP 2 instruction is executed however, the SVC processor will just execute phase 1 and then immediately return to the calling program with all of the registers intact. The calling program can then perform computation or whatever up until it needs the results of the I/O operation that had been started. To complete phases 2 and 3 of the SVC, a TRAP 3 instruction WITHOUT an ID number is executed. If the I/O operation has indeed completed, the return arguments will be immediately loaded into the registers and a return made. If the I/O operation has not yet completed, the TRAP 3 processor will wait for it to complete, load the return arguments, and return. Thus the sequence: TRAP 2 <svc id>, TRAP 3 is exactly equivalent to TRAP 1 <svc id>.

For highly efficient overlap of computation and I/O, arrangements can be made to have the user program interrupted when a currently pending SVC is ready to be completed. When the 6502 is finished with the I/O operation, it will unconditionally set the byte at $000140 non-zero. If the byte at $000150 is non-zero, the user program will be interrupted and the user's service routine will be entered through the vector at $000100. The interrupt service routine can then issue the TRAP 3 to complete the currently pending SVC and perhaps even start another. See section 4.4 for additional information on DMXMON interrupts.

The primary restriction in using "split SVCs" is that only one SVC operation can be in progress at any one time. Thus every SVC that is started with a TRAP 2 must be completed with a TRAP 3 before another SVC may be started with either another TRAP 2 or a TRAP 1. Violation of this restriction will cause DMXMON to halt the user program and print an error message.

4.4 INTERRUPTS

For programs that need them and programmers that can use them, DMXMON has support for handling interrupts. This support is provided without sacrificing programming ease in non-interrupt applications. Since the Datamover is really a slave processor with all of its I/O handled by the 6502, there are no hardware I/O interrupts as such. The 6502 portion of DMXMON however can generate interrupts under certain conditions and the 68000 portion of DMXMON can direct these interrupts to service routines in the user program. Interrupts generated internally by the 68000 (such as divison by zero) are passed directly to the user's service routine (if one is provided) or to DMXMON's error processor which will print complete diagnostic information.

4.4.1 68000 Hardware Interrupts

The MC68000 microprocessor has perhaps more internally generated interrupt sources than any other microprocessor. Most of these refer to error conditions such as zero divide, addressing error, illegal instruction, etc. When DMXMON is started or a RESET command is entered, all of these interrupt vectors are set to point to error message routines. Thus if a divison by zero is attempted and its vector has not been changed, DMXMON will print: "NO USER VECTOR FOR ZERO DIVIDE" and then print the register contents as an aid in locating the error. If the programmer wishes to handle such interrupts himself, the corresponding vector should be changed to point to the programmer's service routine instead.

Because of the Datamover's internal design, many of the 68000's interrupts cannot occur. The table below lists all of the interrupts that can occur, their vector addresses, and what the vector points to by default:

VECTOR	CAUSE	POINTS TO
000004	RESET	DMXMON Initialization routine
00000C	Address error	¥¥¥ADDRESS ERROR¥¥¥
000010	Illegal instruction	¥¥¥ILLEGAL INSTRUCTION¥¥¥
000014	Zero divide	¥¥¥NO USER VECTOR FOR ZERO DIVIDE¥¥¥
000018	CHK instruction	¥¥¥NO USER VECTOR FOR CHK INSTRUCTION¥¥¥
00001C	TRAPV instruction	¥¥¥NO USER VECTOR FOR TRAPV INSTRUCTION¥¥¥
000020	Privilege violation	¥¥¥PRIVILEGE VIOLATION¥¥¥
000024	Trace	¥¥¥NO USER VECTOR FOR TRACE INTERRUPT¥¥¥
000028	Line 1010 emulator	¥¥¥NO USER VECTOR FOR EMULATOR OP CODES¥¥¥
00002C	Line 1111 emulator	¥¥¥NO USER VECTOR FOR EMULATOR OP CODES¥¥¥
000070	Level 4 autovector	DMXMON maskable interrupt service routine
00007C	Level 7 autovector	DMXMON console interrupt service routine
000080	TRAP 0 instruction	DMXMON breakpoint processor
000084	TRAP 1 instruction	DMXMON start-wait-complete SVC processor
000088	TRAP 2 instruction	DMXMON start only SVC processor
00008C	TRAP 3 instruction	DMXMON wait-complete SVC processor
000090 - 0000BF	TRAP 4-15	¥¥¥NO USER VECTOR FOR TRAP INSTRUCTION¥¥¥

Those vectors that point to error messages can be freely changed by the user program to point to the user's error processor. The vectors that point to DMXMON functional routines should not be changed unless there is an overwhelming reason to do so. The vectors for interrupts that cannot occur may either be unused or may be used by DMXMON for system storage. See the detailed memory map in section 5.1 for a complete listing of memory usage by DMXMON.

4.4.2 DMXMON GENERATED INTERRUPTS

The interrupts most likely to be of interest to the programmer are those generated by DMXMON itself. Each of these interrupts has a request flag, an enable flag, and a vector, all stored in low 68000 memory. After reset or on startup, the request flags and enable flags (one byte each) are zeroed and the vectors are set to point to an error message. When a condition that can cause a DMXMON interrupt occurs, the corresponding request flag is unconditionally set to $FF. If the corresponding enable flag is zero, thats all that happens. If the enable flag is non-zero, then an actual interrupt occurs and the corresponding vector points to the user's service routine. The user's service routine is expected perform its function and then return with an RTS instruction (not an RTE!). DMXMON's interrupt dispatcher then zeroes the request flag to indicate that the interrupt has been serviced.

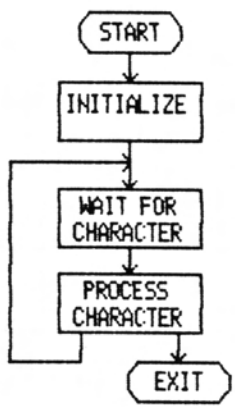
Currently, four DMXMON interrupts are defined and there is space for 12 more. The table below summarizes these:

REQ BYTE	ENAB BYTE	VECTOR	DESCRIPTION
000140	000150	000100	Currently pending SVC has completed
000141	000151	000104	Automatic screen refresh cycle complete
000142	000152	000108	Interrupting keyboard character available
000143	000153	00010C	Timer interrupt
000144	000154	000110	
-	-	-	Unassigned.
00014F	00015F	00013F	

When more than 1 request flag is active, the one corresponding to the lowest address in the table above is checked and acted upon first. Technically, all of these interrupts are on the same level however so the service routine for one cannot be itself interrupted.

4.5 THE INTERRUPTING KEYBOARD

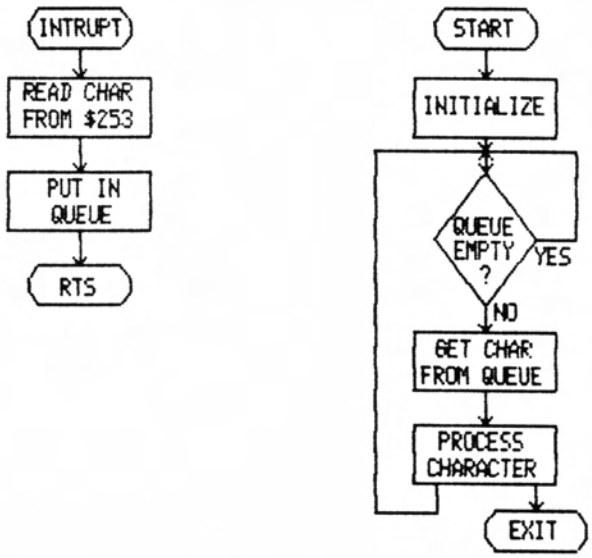
To facilitate the writing of "human engineered" highly interactive programs such as word processors, DMXMON has provisions for an interrupting "background" keyboard. With the normal DMXMON (and CODOS) keyboard routines, when the program needs keyboard input, it calls a keyboard routine (or executes a keyboard SVC) which waits for the operator to press a key. When a key is pressed, the keyboard routine returns and the calling program processes the character. After the character is processed, the keyboard routine is called again for the next character. This process is represented in the flowchart drawing below:



The usual problem with this simple arrangement is that the keyboard is unresponsive while the character is being processed. Normally this is no problem when the processing is relatively simple and only main memory is involved because the processing time is only a few milliseconds. There is also no problem when program communications with the user is of the "command-response" type because the time consuming operations (computation or accessing the disk) only occur when the operator expects to wait after the command is "entered" by pressing Return.

In a free format editing type of program, however, time consuming operations such as moving memory around, scrolling or updating the screen, or accessing the disk may occur at any time. If the operator is busily typing away at one of these times, keystrokes will not be seen by the program and consequently will be lost. Use of the interrupting keyboard facilities of DMXMON is a solution to this problem.

Naturally, program organization is somewhat different (and slightly more complex) when the interrupting keyboard is used. Typical organization is as two "tasks" that are linked together by a first-in-first-out queue as in the diagram below:



Normally, the main program on the right is executing. Assuming that the interrupting keyboard is enabled, when a keystroke is entered, the routine at the left (INTRUPT) immediately gains control through the keyboard interrupt vector arbitrated by DMXMON. This routine reads the ASCII character just entered and stores it in the queue. It then does a Return from Subroutine (NOT Return from Exception!) to indicate that it is finished. The DMXMON interrupt dispatcher clears the interrupt request and then returns control to the main program on the right at the point of interruption.

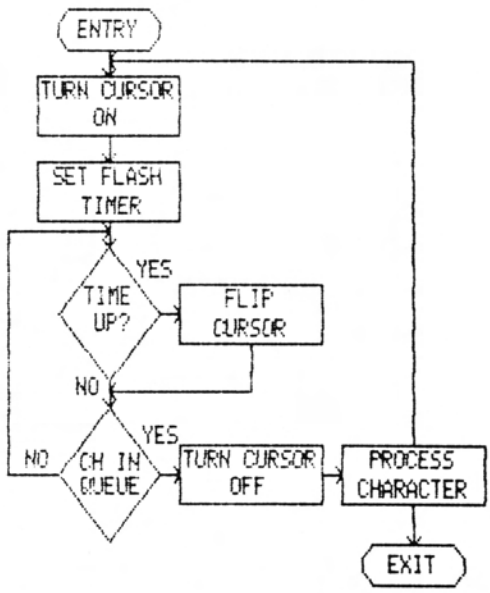
The main program is basically organized just like the simple "wait on keystroke" version described earlier. The only difference is that the block marked "WAIT FOR CHARACTER" is replaced with 2 blocks that look at the queue instead. If the queue is empty, it loops around and looks again. If there is a character (or several characters) in the queue, it is withdrawn and acted upon. After the character is processed, the queue test loop is re-entered. Regardless of how long the "Process Character" block takes, the interrupt service routine will continue to gain control on every keystroke entered so that none are lost. If the process block takes a long time, it is entirely possible for several keystrokes to be "backed up" in the queue. They all will be processed as quickly as possible when the time consuming operation is complete.

Note that this arrangement is much more flexible than the "intelligent keyboards" offered on some computers because in some cases strict queue operation may be undesirable. For example, an error condition may arise as the result of processing a character. At that point, the queue probably should be cleared of any characters that may have backed up because they may represent an inappropriate response to the error condition. Another example would be in an editing program where some keystrokes may have a destructive effect or depend for their effectiveness on hand-eye coordination of the operator (such as moving the cursor). Such characters probably should not be queued so that the display

does not "fall behind" what is being entered. With the interrupting keyboard arrangement, such situations (and perhaps others) are completely under the control of the programmer. A separate intelligent keyboard may not be able to deal with such situations or may be inflexible in handling them.

Whereas the regular "wait for keystroke" SVCs will display a flashing cursor on the screen while waiting and some even display the characters entered, the interrupting keyboard acts independently of anything else and therefore does not interact with the display in any way. This means that the user program is responsible for flashing the cursor and displaying the character. Fortunately, several SVCs are provided to make this task a simple one.

The flowchart below illustrates the typical programming necessary to display a flashing cursor and echo keystrokes to the screen when the interrupting keyboard is used:



Structurally, the flowchart is the same as before. The interrupt service routine is exactly the same so it is not shown. Immediately before the loop that waits for a character to appear in the queue, the text cursor is turned on with an ONTCR SVC. Also, the "flash timer" is started with a CLKON SVC. While in the wait-on-queue loop, the timer expired interrupt request flag at $000143 is tested as well as the queue empty condition (actual interrupts from the timer are not enabled). When the timer expires, the cursor is flipped with a FLPTCR SVC. The cursor continues to be flipped at a rate determined by the timeout value set earlier until a character is found in the queue. Before the character is processed, the text cursor must be turned off, otherwise the reverse video square will remain on the screen if character processing results in cursor movement. The character is then processed and the program loops back to get the next character. For absolute maximum efficiency you may want to check the queue once before the cursor is turned on to avoid the cursor overhead when there is a backup of characters in the queue.

The example program in section 6.3 illustrates how the interrupting keyboard and cursor control SVCs can be used. The program object file is called EXAMPLE2.6 and can be run by starting DMXMON and then entering that file name. The application is just simply typing text onto the screen. To escape the program and return to DMXMON, enter a ctrl/Z. You can check the queueing action by entering a bunch of cursor-downs (which cause time consuming scrolls if the cursor is at the bottom of the screen) followed by some text.

5. APPENDIX

5.1 MEMORY MAPS

5.1.1 MEMORY USED BY 6502 DMXMON

The table below gives the locations in 6502 memory that are used by DMXMON. Locations not listed may be freely used by 6502 utilities and other programs executed through DMXMON or by user written subroutines called from a 68000 program.

ADDRESS RANGE	USED FOR
0055 - 00AF	Zero page variable storage
0700 - 0AFF	General variable storage
0B00 - 3941	Code for 6502 portion of DMXMON
3942 - 4941	Copy of code for 68000 portion of DMXMON
1FC00 -1FC38	Virtual screen copy routines (Bank 1)

5.1.2 MEMORY USED BY 68000 DMXMON

The table below gives the locations in 68000 memory that are used by DMXMON. Locations listed as -unused- may be utilized by user programs. It is recommended however that user programs confine their memory usage to locations above $001000. Note that the RESET command will restore all 68000 memory from $000000 to $000FFF to the state it was immediately before DMXMON was entered from CODOS.

ADDRESS RANGE	USED FOR
000000 - 000003	Initial stack pointer after Reset
000004 - 000007	Initial program counter after Reset
000008 - 00000B	Hardware bus error vector (should not be possible)
00000C - 00000F	Address error vector
000010 - 000013	Illegal instruction error vector
000014 - 000017	Zero divide error vector
000018 - 00001B	Check against bounds vector
00001C - 00001F	TRAPV instruction vector
000020 - 000023	Privilege violation vector
000024 - 000027	Trace interrupt vector
000028 - 00002B	1010 emulator vector
00002C - 00002F	1111 emulator vector
000030 - 00005F	-unused-
000060 - 000063	Spurious interrupt vector (should not be possible)
000064 - 00006F	Autovectors 1-3 (should not be possible)
000070 - 000073	Level 4 autovector to DMXMON interrupt dispatcher
000074 - 00007B	Autovectors 5-6 (should not be possible)
00007C - 00007F	Level 7 autovector to DMXMON console interrupt handler
000080 - 000083	TRAP 0 vector to DMXMON breakpoint processor
000084 - 00008F	TRAP 1-3 vectors to DMXMON SVC processor
000090 - 0000BF	TRAP 4-15 vectors
0000C0 - 0000FF	-unused-
000100 - 000103	SVC operation complete interrupt vector
000104 - 000107	Automatic screen update completed interrupt vector
000108 - 00010B	Keyboard interrupt vector
00010C - 00010F	Clock interrupt vector
000110 - 00013F	Unused DMXMON arbitrated interrupt vectors

000140	SVC operation complete interrupt request flag
000141	Automatic screen update completed interrupt request flag
000142	Keyboard interrupt request flag
000143	Clock interrupt request flag
000144 - 00014F	Unused DMXMON arbitrated interrupt request flags
000150	SVC operation complete interrupt enable flag
000151	Automatic screen update completed interrupt enable flag
000152	Keyboard interrupt enable flag
000153	Clock interrupt enable flag
000154 - 00015F	Unused DMXMON arbitrated interrupt enable flags
000160 - 0001FF	-unused-
000200 - 00021F	Save area for data registers
000220 - 00023F	Save area for address registers & system stack pointer
000240 - 000243	Save area for user stack pointer
000244 - 000247	Save area for program counter
000248 - 000249	Save area for status register
00024A - 00024F	-unused-
000250 - 000252	DMXMON internal use flags
000253	Character entered from the interrupting keyboard
000254	DMXMON internal use flag
000255 - 00025B	-unused-
00025C - 00025F	DMXMON internal use
000260 - 000263	Address of system input line buffer set by SVCDFB
000264 - 000267	Address of system output line buffer set by SVCDFB
000268 - 00026B	32 bit clock controlled by CLKON and CLKOFF SVCs
00026C - 00027F	-unused-
000280 - 00029F	Arguments to 6502 SVC processor
0002A0 - 0002BF	Arguments from 6502 SVC processor
0002C0 - 0004FF	Default system stack
000500 - 0005FF	Default input line buffer
000600 - 0006FF	Default output line buffer
000700 - 000DC9	DMXMON program code
000DCA - 000FFF	Unused, reserved for DMXMON expansion
001000 - 03FFFF	-unused- available for user programs (standard Datamover)
040000 - 0FFFFF	-unused- available for user programs (expansion memory)
100000 - 10FFFF	Decoded as on-board I/O
110000 - 11FFFF	Decoded as off-board I/O
120000 - 1FFFFF	Not decoded
200000 - FFFFFF	Seven duplicates of 000000 - 1FFFFF

5.2 ERROR MESSAGES

Listed below are all of the error messages that can originate from DMXMON. In addition to these, CODOS may sometimes print its own message which can be recognized because CODOS errors are numbered.

5.2.1 CONSOLE MODE MESSAGES

<FROM> ARGUMENT MISSING OR ILLEGAL
<TO> ARGUMENT MISSING OR ILLEGAL
<DEST> ARGUMENT MISSING OR ILLEGAL
<VALUE> ARGUMENT MISSING OR ILLEGAL
<CHANNEL> ARGUMENT MISSING OR ILLEGAL
<DEVICE> ARGUMENT MISSING OR ILLEGAL
<FILENAME> ARGUMENT MISSING OR ILLEGAL
<ENTRY> ARGUMENT MISSING OR ILLEGAL
<FROM> ARGUMENT GREATER THAN <TO>
NEW FILENAME ALREADY EXISTS
FILE NOT FOUND
<VALUE> ARGUMENT NEGATIVE OR > $FF
ARITHMETIC OVERFLOW
STRING DELIMITER MISSING OR ILLEGAL
MEMORY VERIFY FAILED AFTER SET/FILL
DRIVE NOT OPEN
FILE IS LOCKED
NOT A ".6" LOADABLE FILE
ILLEGAL REGISTER ID
ILLEGAL BREAKPOINT ADDRESS (MUST BE EVEN)
TOO MANY BREAKPOINTS DEFINED
ARGUMENT MISSING OR ILLEGAL

5.2.2 EXECUTION MODE ERRORS

When an unrecoverable error occurs during execution mode, one of the error messages listed below is printed followed by a printout of the register contents. The "UNSUCCESSFUL SVC EXECUTION" message may be preceeded by a CODOS error message which identifies the nature of the problem. In either case, DMXMON enters console mode after an execution mode error.

¥¥¥68000 SYSTEM CRASH - UNRECOGNIZED COMPLETION CODE¥¥¥ - Is generally caused by a 68000 program that has run wild and wiped out memory. A RESET command should be issued to reload 68000 DMXMON.

-CONSOLE INTERRUPT- - The INT key was pressed during execution mode.

¥¥¥ADDRESS ERROR¥¥¥ - Typically caused by trying to address a word or long word operand at an odd address.

¥¥¥ILLEGAL INSTRUCTION¥¥¥ - The 68000 CPU has tried to execute an undefined op code.

¥¥¥PRIVILEGE VIOLATION¥¥¥ - The user program has changed the privilege level to User and then tried to execute a privileged instruction.

¥¥¥NO USER VECTOR FOR TRACE INTERRUPT¥¥¥ - A trace interrupt occured but its vector had never been defined.

¥¥¥NO USER VECTOR FOR ZERO DIVIDE INTERRUPT¥¥¥ - A divison by zero was attempted and the zero divide vector had never been defined.

¥¥¥NO USER VECTOR FOR CHK INSTRUCTION¥¥¥ - A CHK instruction that failed was executed and the CHK instruction vector had never been defined.

¥¥¥NO USER VECTOR FOR TRAPV INSTRUCTION¥¥¥ - A TRAPV instruction that failed was executed and the TRAPV instruction vector had never been defined.

¥¥¥NO USER VECTOR FOR EMULATOR OP CODES¥¥¥ - The 68000 CPU has tried to execute an undefined op code whose first hex digit is either $A or $F and the emulator opcode vector had never been defined.

¥¥¥NO USER VECTOR FOR USER TRAP INSTRUCTION¥¥¥ - A TRAP 4-15 instruction was executed and its vector had never been defined.

¥¥¥NO USER VECTOR FOR AUTOVECTOR INTERRUPT¥¥¥ - A hardware failure has caused an autovector interrupt.

¥¥¥NO USER VECTOR FOR DMXMON INTERRUPT¥¥¥ - One of the DMXMON arbitrated interrupts has been enabled but the associated vector was never defined.

¥¥¥UNIMPLEMENTED 68000 SVC¥¥¥ - An SVC with an undefined ID was attempted.

-BREAKPOINT- - A breakpoint was encountered during execution.

¥¥¥NEW SVC ATTEMPTED BEFORE PREVIOUS SVC COMPLETE¥¥¥ - When using overlapped I/O and computation, the current SVC was not completed with a TRAP 3 before another one was started with a TRAP 1 or TRAP 2.

¥¥¥UNSUCCESSFUL SVC EXECUTION¥¥¥ - Due to an error condition, an SVC could not be completed successfully. There will typically be a CODOS message before this message.

¥¥¥68000 SYSTEM CRASH - HARDWARE BUS ERROR¥¥¥ - A hardware failure has caused the 68000 to recognize a bus error.

5.3 LOADABLE FILE FORMAT

Datamover memory images saved by DMXMON with the SAVE command have a default extension of .6 and use the following format for each memory block stored:

BYTE NUMBER	CONTENTS	DESCRIPTION
0	$B6='6'+$80	Header byte to indicate 68000 loadable file
1	$41='A'	Absolute block code
2-3	$0020	Size of the block header = 32 bytes
4-7	$00	Reserved for future use
8-11	<from>	Load address for the block
12-15	<to>-<from>+1	Size of the block
16-19	<entry>	Entry point address (meaningful on first block)
20-31	$00	Reserved for future use
32-(Size+31)	<data>	Saved Datamover memory image for the block
.....		Additional blocks, if present

Multi-byte numbers are stored in the file least significant byte first. If multiple blocks are stored in the file, the above format is merely repeated as many times as necessary. The GET command continues loading until end-of-file is encountered.

6. SAMPLE PROGRAMS

On the DMXMON distribution disk are three sample programs that use DMXMON SVCs extensively. IMAGETRANS illustrates general SVC usage, particularly those associated with operator dialog through the console and disk file manipulation. HATCOMP illustrates use of the graphic SVCs for creating drawings directly on the MTU-130 screen and use of the "virtual screen" feature of DMXMON. INTKBDEMO illustrates an interrupt driven 68000 program using the optional interrupting keyboard support features of DMXMON. The main purpose of these programs is to illustrate how SVCs are used in typical, non-trivial programming situations. They also contain some generally useful 68000 subroutines including decimal number decoding, 32 bit square root, and table-lookup sine and cosine.

Both the source file (<name>.A) and object file (<name>.6) for each program is included on the DMXMON distribution disk. The source files are written so that they can be assembled using the MTU ASM or MACASM 6502 assemblers; a 68000 assembler is not required. Use the following procedure if you wish to alter one of the programs and re-assemble it:

1. Be sure the files OP68000.A and SVC68000.A are on the disk along with the source.

2. After making the desired changes with the editor, assemble the program in the usual manner using either ASM or MACASM.

3. While in CODOS, load the object file into Datamover memory with a GET command (a .BANK 2 statement is included in the source so that it will automatically be loaded into Bank 2 which is low Datamover memory).

4. Execute a GETLOC command to determine where the program has loaded.

5. Start up DMXMON by entering DMXMON and a carriage return.

6. SAVE the program from DMXMON so that it will be in DMXMON loadable format instead of CODOS loadable format. The default file extension is .6.

7. You may delete the CODOS loadable object file produced by the assembler now if you wish.

8. The re-assembled program may be executed from DMXMON by simply typing its name.

6.1 IMAGETRANS

This program illustrates use of the inline message, line input, record input, record output, file assignment, and other character oriented and file handling SVCs. It incorporates extensive error checking as an example of how that is performed as well. Its actual function is to convert a screen dump disk file which is A pixels wide and B pixels high into a similar file B pixels wide and A pixels high. This may actually be useful for rotating images that will be printed by the MTU word processor, WORDPIC.

To start the program, just type its name in response to a DMXMON prompt. It will clear the screen, identify itself, and ask for a file name. The file must be a memory image, either of 6502 memory or of 68000 memory, must be on a non-write protected disk, and must be unlocked. All of these requirements are checked by the program and an appropriate message printed if there is a problem. It then asks for the dimensions of the image stored in the file. Each dimension should be a multiple of 8. Following this, the entire file is read into memory rowwise and then rewritten columnwise. A 1024x1024 image file (128K bytes) requires about 2 minutes to transpose.

6.2 HATCOMP

This program illustrates the use of graphics oriented SVCs. It both draws directly on the MTU-130 screen using SMOVE, SDRAW, and SVEC SVCs, and also draws on a "virtual screen" in 68000 memory and uses the window copy SVCs to transfer to the '130 screen. It also illustrates a number of programming and operator interaction techniques and contains some useful math oriented subroutines. Its actual function is to compute the famous MTU HAT image on a 1024 by 1024 grid and allow the operator to interactively move a smaller window around on this grid to view portions of the image. With a suitable printer (such as the NEC PC-4023) and program (such as WORDPIC's PRINT program), the image may be printed in its entirety on paper.

To run the program, simply type its name, HATCOMP, in repsonse to a DMXMON prompt. It will clear the MTU-130 screen, draw two rectangles at the left, and immediately begin computation of the hat image. The larger rectangle represents the full 1024x1024 virtual image plane (1/8 size) and the smaller rectangle represents the currently visible window. Even while computation is taking place, you may move the window around with the cursor arrow keys. Movement is in steps of 8 pixels. If the shift key is held down, movement is in steps of 64 pixels. After approximately 10 minutes, the completed image will be copied into the large rectangle at the left for reference. The window may still be moved around. To halt the program, press the INT key. To save the image on disk for printing (or processing by the IMAGETRANS program), SAVE memory from 20000 through 3FFFF while in DMXMON.

6.3 INTKBDEMO

This program illustrates programming techniques for the interrupting keyboard feature of DMXMON. The program itself simply displays whatever is typed but keyboard input is queued and keyboard scanning is done with an "N-key rollover" algorithm. This means that keystrokes will never be lost no matter what the screen is doing or how fast the operator types or how many keys (within reason) are down simultaneously.

To start the program, type its name, INTKBDEMO, in response to a DMXMON prompt. The screen will be cleared and the cursor will appear in the home position flashing faster than normal. You can test the queueing function by pressing a number of keys at once (such as all of the number keys) and noting that they are all recognized although the exact order may be unpredictable. You may also try a number of cursor downs to queue up a few screen scrolls and note that subsequent typed input is not lost although there will be a delay before it is displayed. The program may be stopped by entering cntl/Z. If it is stopped with the INT key, the interrupting keyboard will still be enabled and you will get <u>two</u> clicks for each keystroke! Enter a RESET command to clear this condition.