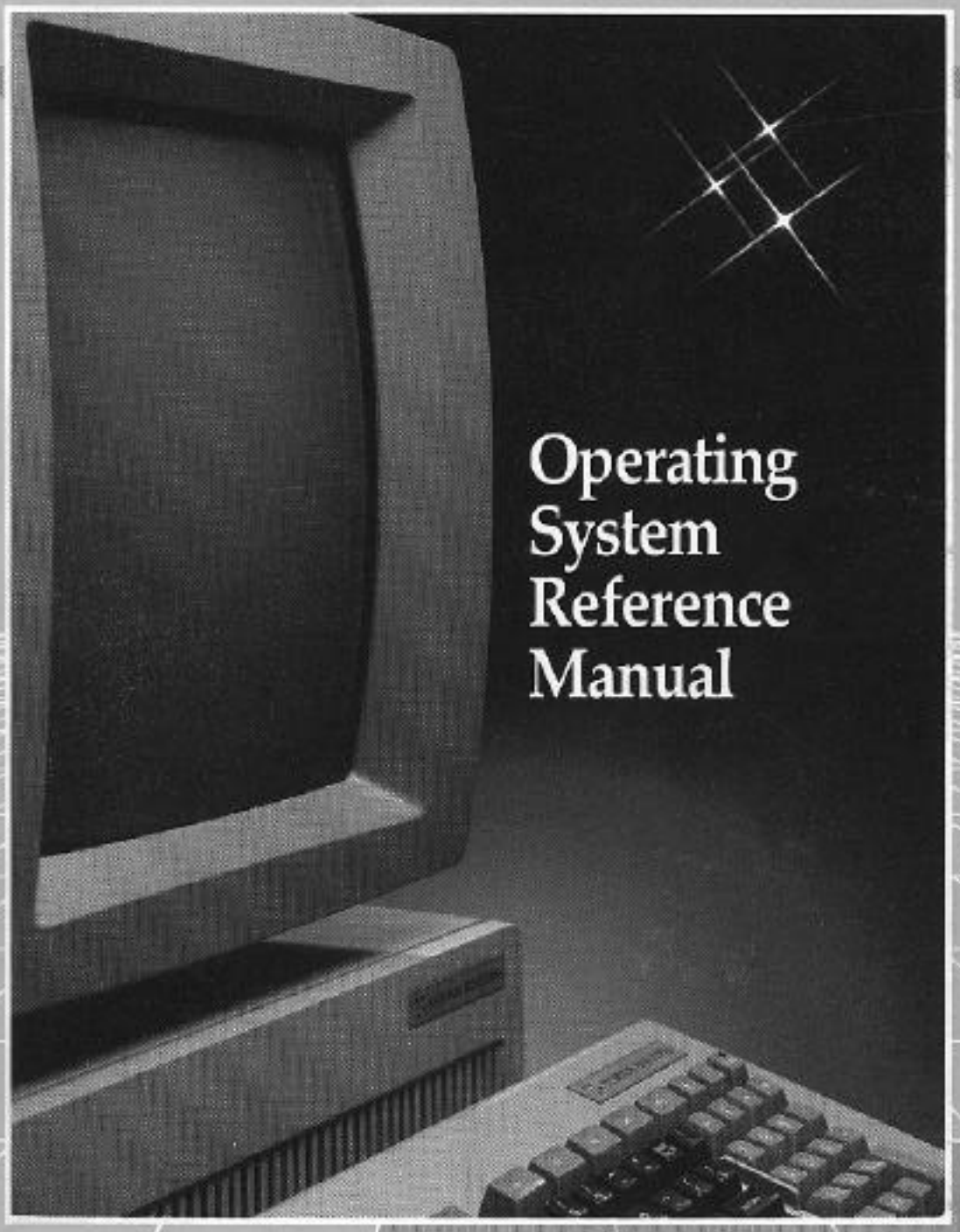
THE CORVUS CONCEPT



Operating System Reference Manual

THE CORVUS CONCEPT

OPERATING SYSTEM REFERENCE MANUAL

Part No.: 7100-02825
Document No.: CCC/36-33/1.2
Release Date: September, 1983

Corvus Concept™ is a trademark of Corvus Systems, Inc.

Table of Contents

General Information 1

Topics covered in this chapter include:

* CCOS devices and units
* CCOS internal data structures
* The system communications area (SYSCOM)
* The system call vector
* Device directories
* IORESULT codes

Devices, Units, Volumes, and Drivers

The primary function of an operating system is to control the interaction between the computer and its various peripherals. The actual, physical peripherals, such as printers and disk drives, are called devices. The operating system may view the device as one or more logical peripherals or volumes. Modems and printers are examples of devices that the operating system treats as a single volume. Hard disks are generally treated as multiple volumes.

CCOS uses the unit mechanism to assign logical volumes to physical devices. At boot time, each volume is assigned a specific unit number that the operating system uses to communicate with it. This unit number may vary depending upon the system's configuration.

Drivers are the software that implement the unit mechanism. A driver has two purposes: to transform generic operating system commands into device-specific actions, and to perform whatever format conversion is necessary when passing data between the computer and the peripheral. CCOS recognizes two types of devices: block devices, which communicate in 512-byte chunks, and character devices, which communicate on a character-by-character basis.

File and Volume Names

CCOS volume names may be up to seven characters long, and can contain only letters, digits, and periods. A volume name can start with letters or digits, but not periods. File names may start with letters, digits, or periods, and may be up to 15 characters long.

Many CCOS file names use the period character to mark extensions to the file name. This period character is included as one of the 15 allowable characters.

CCOS uses three standard extensions in file names:

.TEXT - A text file is human readable text, such as file produced when an EdWord wordpad is saved.

.I - Intermediate code files produced by various compilers. These files are normally deleted automatically as part of compiler operation.

.OBJ - The end result of a compilation or assembly. These files are machine executable, although most will require an additional linking step.

User files may have any extensions desired.

Temporary files

Temporary files are normally not seen during standard CCOS operation. They are most often encountered after a system crash or power failure during a compilation or assembly. Temporary files are easily indentified by their scrambled datestamps- a directory listing will show a file with a ??? replacing the month specifier, and a year of 100. When a compilation or assembly aborts with a

```
Can't open intermediate code file
```

message, it's generally an indication that a previous operation has left a temporary file that needs to be erased.

Temporary files can only be deleted with the [DletTemp] command from the file manager. If necessary, a temporary file can be made permanent by changing the date record of the file entry to a legitimate value with a user-written program.

The Dispatcher

After CCOS is booted, the file CC.DISPAT is automatically executed. This program is called the dispatcher, and is the primary user interface to CCOS.

All CCOS utility and applications programs are run from the dispatcher. When other programs are executed, the operation of the dispatcher is suspended, and its state saved.

In some situations it may be useful to exit the dispatcher and issue commands directly to the operating system. The dispatcher is exited by pressing

```
[ExecFile] % <Return>
```

The function key labels will disappear from the bottom of the screen, and the CCOS % prompt will appear in the system window. The dispatcher, however, is still in memory, and typing control-D will cause the system to return control to it.

Units and Volumes

CCOS communicates with peripherals and devices through the unit mechanism. There are two types of units: character units and block units. As their names imply, character units work with I/O on a character-by-character basis, while block

units deal with 512-byte blocks. Block units are generally disk drives, tape drives, or other mass storage devices.

Devices on the system may be refered to either by their volume name or their unit number, depending upon the environment. From the dispatcher, volume names must be used. From a program, use of the file I/O intrinsics requires volume names, while use of the UNIT I/O intrinsics requires unit numbers.

CCOS assigns unit numbers to devices on the system during the boot process. While physical devices such as the keyboard, display, and RS-232 ports will always be assigned the same unit number, logical devices such as disk volumes may be assigned different unit numbers, depending on the order in which they are mounted. Therefore, it's generally a good idea to use volume names when referencing disks from within programs. Two special forms of the [ListVol] function are available:

! [ListVol] and / [ListVol]

These can be issued from either the dispatcher or the file manager. The first form shows all current devices and volumes, along with their unit numbers, slot numbers, size in blocks, and other information. The second form show more abbreviated information consisting of just the unit numbers, volume type, and size in blocks.

All volume names are preceeded by a slash character. For example, the string /WORK refers to the volume named WORK, while /WORK/GRAPHICS is the PATHNAME for the file GRAPHICS on the volume WORK.

The standard CCOS unit numbers and their assignments are:

Unit Number	Name	Description
0	/Null	Null device. May be written to indefinitely; when read, an end-of-file is returned. UNITBUSY will always return a FALSE.
1	/Console	The system keyboard and screen, with echo.
2	/Systerm	Same as /console, but without echo.
3	-----	Unassigned and available for

		user devices.
4	/CCSYS	The root volume, or volume from which the system was booted. On the Concept, this is generally /CCSYS.
5	-----	A user volume. The name depends on the volume assigned to that unit. If a volume is assigned, CCOS makes it the default volume.
6	/printer	The system printer, if one is available.
7	/remin	Not currently used by CCOS; available for user devices.
8	/remout	Not currently used by CCOS; available for user devices.
9-29	-----	User devices; generally disk volumes.
30	/SLOTIO	General slot I/O routines.
31	/DTACOM1	RS-232 port 1 driver.
32	/DTACOM2	RS-232 port 2 driver.
33	/OMNINET	Omninet port driver.
34	/TIMER	Timer driver used by CCOS.
35	/KYBD	Keyboard driver.
36	/DISPLAY	Horizontal or vertical display driver.

Floppy disks, if present, will be mounted starting on unit 9. Unless specifically mounted to a certain unit (via the mount manager utility), hard disk volumes start usually mounting at unit 10, or the first unit number available after all floppies have been mounted.

The dispatcher recognizes two special volumes: the boot volume and the default volume. The root volume is the volume the system was booted from, and always has a unit number of 4. The default volume is set during a boot to whatever vol-

ume is mounted on unit 5, or the root volume if there is no volume mounted on unit 5. The default volume may be changed wktx the [SetVol] command from the file manager or dispatcher.

Operating System Data Types

BYTES: 8 bit quantities which are interpreted as values in the range -128 to +127.

BLOCKS: A block is a group of 512 bytes, and is the standard unit of disk I/O.

WORDS: Occupy 16 bits, and are equivalent to the Pascal type integer. Words represent signed integers in the range -32768 to 32767 and are always aligned on word boundaries.

LONG WORDS: Long words are 32 bits long and correspond to the Pascal longint data type. Long words are aligned on word boundaries. Long words represent signed integers in the range -2,147,483,648 to 2,147,483,647.

BOOLEANS: A Boolean data type occupies a single byte. A value of 1 represents TRUE; a value of 0 represents FALSE. Other values are not valid Booleans. Packed Booleans occupy one bit each.

POINTERS: A pointer is a long word structure whose contents are a specific address within the 16M address space of the 68000 processor.

The NIL pointer: Pointers, as mentioned above, are long word quantities. The NIL pointer points to nothing and is represented by a value of 0.

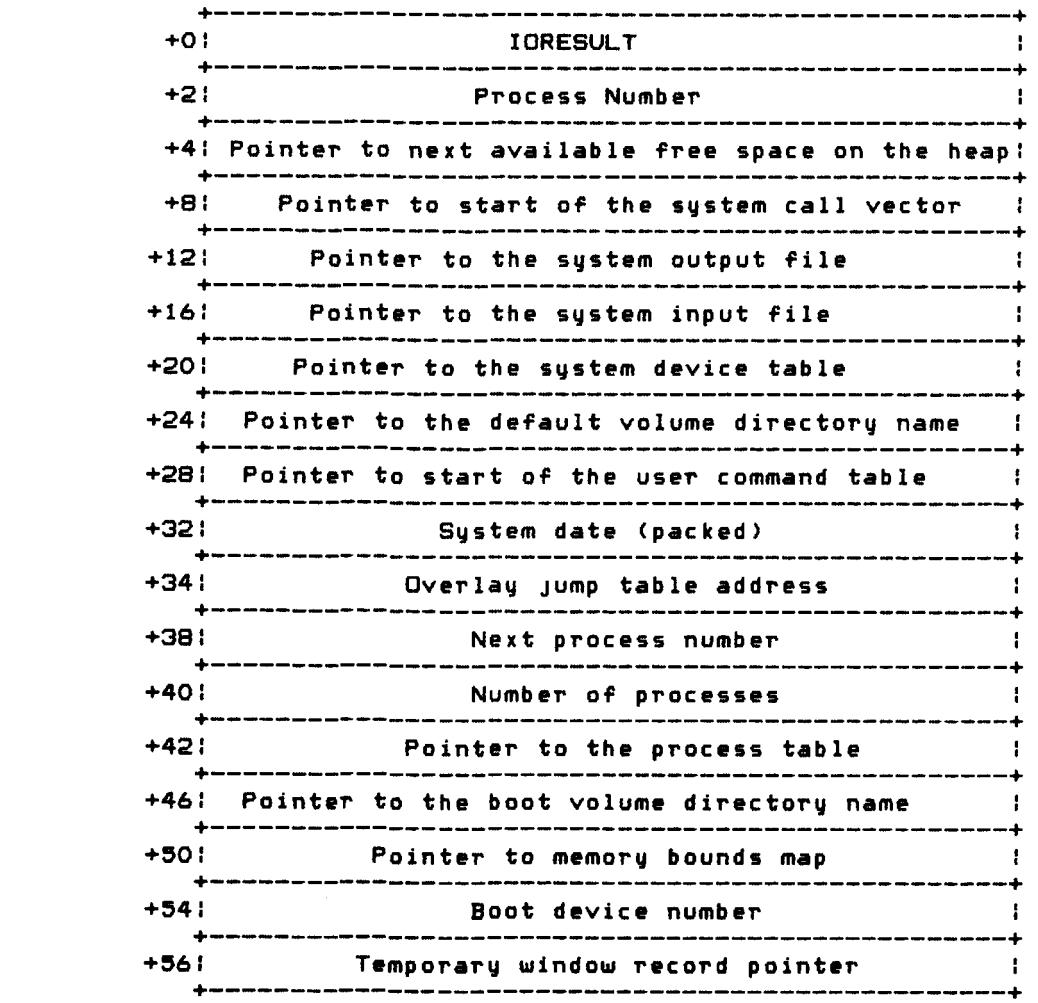
STRINGS: A Pascal data type consisting of a (packed) array of characters with a preceding length byte. Thus, a string has a maximum of 255 elements. The length byte does not count as part of the length of the string. Strings are aligned on word boundaries.

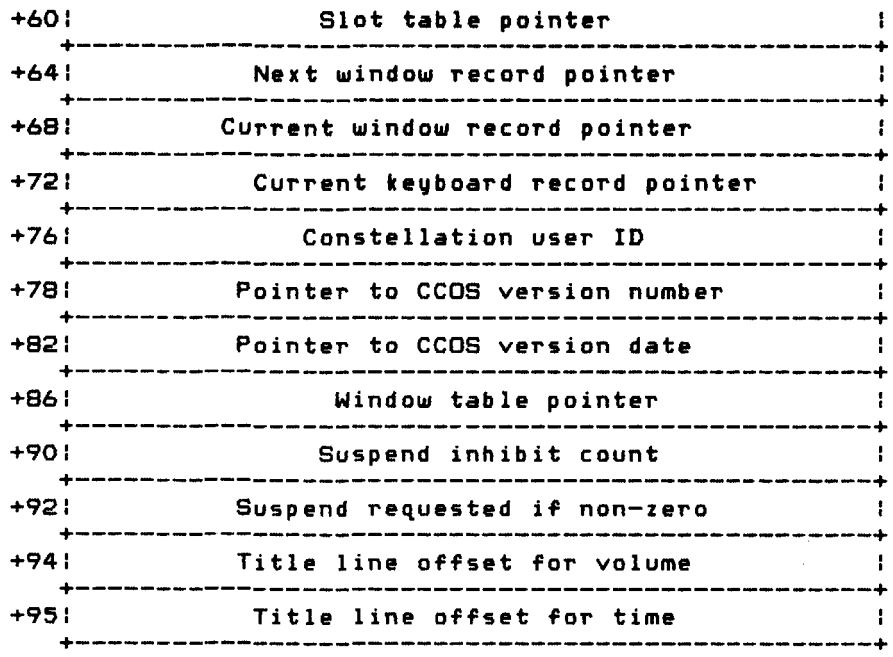
PACKED ARRAY OF CHARACTER: Similar to the string type, but without the preceding length byte. Also aligned on word boundaries.

The System Communications Area

CCOS maintains a system communications area, or SYSCOM, in memory. SYSCOM contains important, global system information that is used by the operating system.

The exact location of SYSCOM varies, but its address is contained in the pointer at $180. Below is a diagram of the data in SYSCOM. The numbers to the left of each field indicate the displacement of the field into SYSCOM, in bytes. Following the diagram are short explanations of each of the fields.





IOResult: A word value which contains a result code after the completion of any I/O process.

Process number: A word value which is the current process number. The dispatcher has a process number of 0. A maximum of ten processes may be in use at any one time.

Free heap: A pointer to the start of free memory available for storage allocation on the heap.

System call vector- A pointer to the system call vector. This is a jump table to the various system routines and is described in further detail in the section SYSCOM: The System Call Vector.

Sysout: A pointer to the standard output file. This is generally the screen. SYSIN and SYSOUT are used by the system for last resort error messages, such as when the system runs out of memory.

Sysin: A pointer to the standard input file, generally the keyboard (handling routine).

System device table: A pointer to the device table. The device table describes each unit number to the system.

Directory name: A pointer to the default directory name.

User table: A pointer to the start of the user command table.

Date record: A packed, one-word record containing the current system date.

Overlay table address: A pointer value pointing to the start of the current process overlay table. Used only when the current process contains overlays; otherwise contains a 0.

Next process number: A word value to be assigned to the next process.

Number of processes: A word value representing the number of processes currently active, including the dispatcher.

Process table address: A pointer to the process table. The process table contains information on the processes currently in the system.

Boot name: A pointer to the directory of the device used to boot the system.

Mem map: A pointer to a table describing the limits of memory available to CCOS on the current hardware.

Bootdev: A word value containing the number of the initial boot device.

Temp window record pointer: A pointer to a window record for a temporary window.

Slot table pointer: A pointer to the slot table.

Next window rec: A pointer to the next window record; valid only when more than one window is in use (CCOS normally keeps 3 windows active).

Current window rec: A pointer to the current window record.

Current keyboard rec: A pointer to the current keyboard character translation tables.

Constellation user ID: A word containing the current user number.

CCOS version number: A pointer to the a string containing the current CCOS version number.

CCOS version date: A pointer to the release date for the current version of CCOS.

Window table: A pointer to the window table.

Suspend inhibit count: Not used.

Suspend requested if not zero: Not used.

Title line offset for volume: Assuming the default 6 by 10 character set, the number of spaces over to print the volume name on the title line at the top of the screen. This will vary depending upon the screen orientation.

Title line offset for date: The number of spaces over to print the date.

Additional information on the various fields in SYSCOM can be found in chapter 6.

Accessing SYSCOM from Pascal

You can easily access SYSCOM from a Pascal program. The following program fragment illustrates the technique:

```
Program Raskin;

Uses {$U /ccutil/os.globals.obj} globals;

Var
 PSys : PSysCom;     { Defined in GLOBALS }


Begin
  PSys := Pointer(SysComRec);   { Defined in GLOBALS }
  WriteLn('Current IORESULT is ',psys^.sioresult)
End.
```

Note that this program assumes the existence of the GLOBALS unit in CCUTIL. The globals unit cannot be used in a program that uses the standard library unit CCdefn since many identifiers are declared in both units. In some cases it will be necessary to extract only the data structures needed (such as SYSCOMREC and FIB declarations) from the globals unit.

SYSCOM: The System Call Vector

All CCOS system calls are made through a table of routine addresses. This table is the system call vector. Each entry in the table is a pointer to the address of the desired routine. The address of the system call vector may be found in the SYSCOM (see previous section). The table layout is as follows:

Offset	Routine Name	Description
0	UNIT WRITE	Direct write to a unit
4	UNIT READ	Direct read from a unit
8	UNIT CLEAR	Reset a unit- flush buffers, if any.
12	UNIT BUSY	Check to see if unit is busy
16	FPUT	Write one record to a file
20	FGET	Get one record from a file
24	FINIT	Initialize a file
28	FOPEN	Open a file
32	FCLOSE	Close a file
36	WRITE CHAR	Write a character to a file
40	READ CHAR	Read a character from a file
44	BLOCK I/O	Block file I/O
48	FSEEK	Position a file to a record
52	NEW	Allocate memory on the heap
56	DISPOSE	Remove allocated memory. This is currently a NOP; use MARK and RELEASE to manage memory
60	MARK	Mark the current top of heap
64	RELEASE	Cut heap back to MARKed pos.
68	MEMAVAIL	Returns memory available for dynamic storage allocation

72	GETDIRNAME	Get current directory name
76	CRACKPATHNAME	Parse a pathname
80	UNITSTATUS	Unit status call
84	LNEW	LONGINT version of NEW
88	LDISPOSE	LONGINT version of DISPOSE
92	CLI	Command line interpreter
96	GETVOLNAMES	Get volume names
100	CHKDIR	Check for valid directory
104	FLPDIR	Flip directory
108	SEARCHDIR	Search directory for filename
112	DELDIRENTRY	Delete directory entry
116	PUTDIR	Write directory
120	UNITINSTALL	Install a unit driver

Calling a System Routine

Most user programs in a high level language will not need to issue direct system calls, since the standard routines supplied in each language (i.e. WRITE and READ in Pascal) will provide all the necessary functions. The only way to call a CCOS routine directly from a high level language is to code an external assembly language routine and link it to the main program.

To call a system routine from an assembly language program, the appropriate parameters for the routine are pushed onto the stack, and a JSR to the appropriate routine address (extracted from the system call vector table) is executed.

Below is an example of a system call to close an open file:

```
PEA     FBUF          ; Push FIB address
CLR.W   -(SP)         ; Close type := NORMAL
MOVE.L  $180.W,A0     ; A0 now = SYSCOM address
MOVE.L  8(A0),A0      ; A0 now = sys call vector address
```

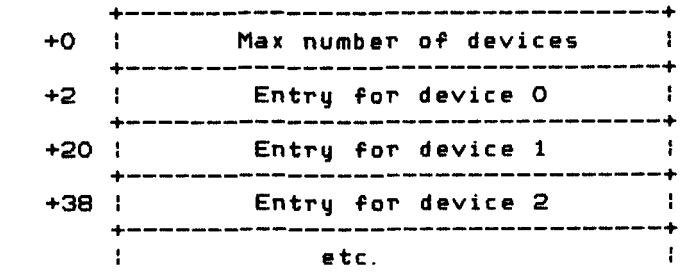
```
MOVE.L 32(A0),A0   ; A0 now = FCLOSE address
JSR    (A0)        ; CCOS call
....               ; CCOS returns here. Remember to
                   ;  check IORESULT!!!
```

The program fragment above assumes that the user has declared an appropriate FIB for the file being opened. The FCLOSE routine expects this address on the stack, which is accomplished by the first instruction. The third, fourth, and fifth instructions locate syscom, then locate the system call vector and extract the address of the FCLOSE routine, leaving it in A0. An indirect JSR then calls the routine. The system call vector should always be used to call system routines since their addresses may change in different releases of CCOS and different memory size machines.

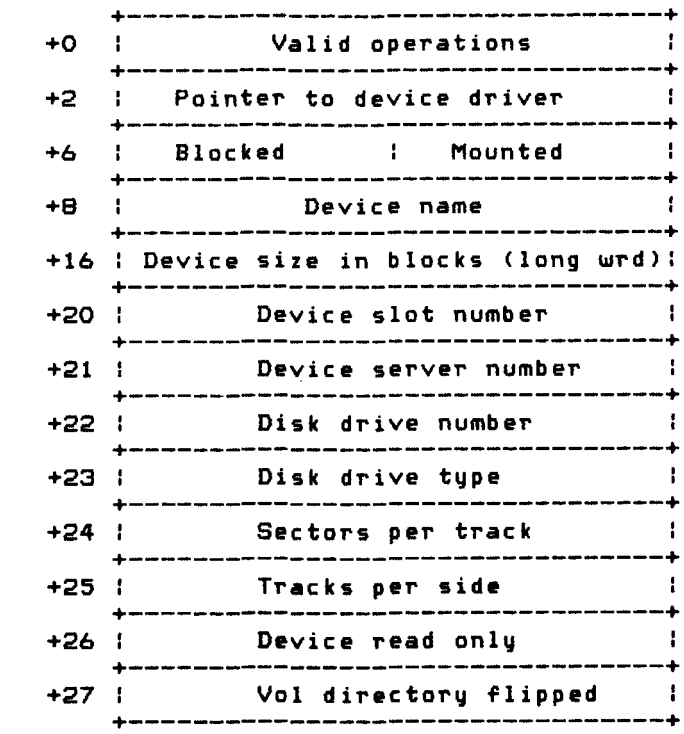
System calls and details on the various parameters are dicussed more fully in chapter 3.

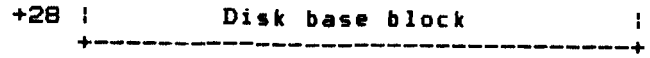
SYSCOM: The Device Table

The device table or unit table contains a list of all devices currently recognized by the operating system. The address of this table is kept in SYSCOM at an offset of 20 bytes. The table structure is:



The first word of the table contains a number representing the maximum number of devices available on the system. Each successive 18 byte entry contains the characteristics of a single device. The format of a device entry is:





Note: For the Revision B 8" floppy disk driver, the "disk base block" parameter becomes "sector size in bytes." Only the lower order word of the parameter is used.

Valid Operations: This is a word quantity whose individual bits specify which operations are possible. The bits used and their definitions are:

0: UNITREAD
1: UNITWRITE
2: UNITCLEAR
3: UNITBUSY
4: UNITSTATUS

Pointer to device driver: A pointer to the entry point for a device driver.

Blocked: A Boolean that indicates (when TRUE) that a device is blocked, such as a disk drive. Non-blocked devices, such as printers, handle I/O on a character by character basis.

Mounted: A Boolean that indicates (when TRUE) that a device is mounted.

Device name: An 8 byte string field containing the name of the device. The first byte is a length byte; the remaining seven bytes contain the actual device name. If no valid media is present (such as might occur when there is no floppy disk in a disk drive), the length byte will be 0.

Device size: A long word quantity indicating the number of 512 byte blocks residing on the device. This is applicable only to blocked devices. For unblocked devices such as printers or modems, this is set to 32,767.

Device slot: Applicable only for devices attached to the Concept via the 50-conductor I/O slots. A byte which contains the slot number the device resides in. Set to 5 for Omninet devices.

Device server: A byte quantity containing the device's server number. Applicable only to network systems.

Disk drive: A byte containing the number of the disk drive the volume resides on.

Sectors per track: For floppy disks, a byte containing the number of sectors per track.

Tracks per side: For floppy disks, a byte containing the number of tracks per side.

Device read only: A Boolean quantity set to TRUE if the device is read only.

Vol directory flipped: A Boolean quantity set to TRUE if the device directory is byte flipped. Applicable only to blocked devices. A flipped directory has its integer fields stored in a format of low order byte first, instead of the CCOS standard of high order byte first.

Disk base block: A long integer containing the starting block number of the device on a disk. Used for volume offsets on a hard disk. For floppy drives, this field contains the sector size in bytes in its low order word. woerds

Accessing the device table from Pascal

The following program fragment shows how the information in the device table may be accessed from a Pascal program:

```
Program Raskin;

Uses {$U /ccutil/os.globals.obj} globals;

Var
 PSys : PSysCom;
 PTab : PDevTable;

Begin
  PSys := Pointer(PSysCom);
  PTab := PSys^.SysDevTab;
```

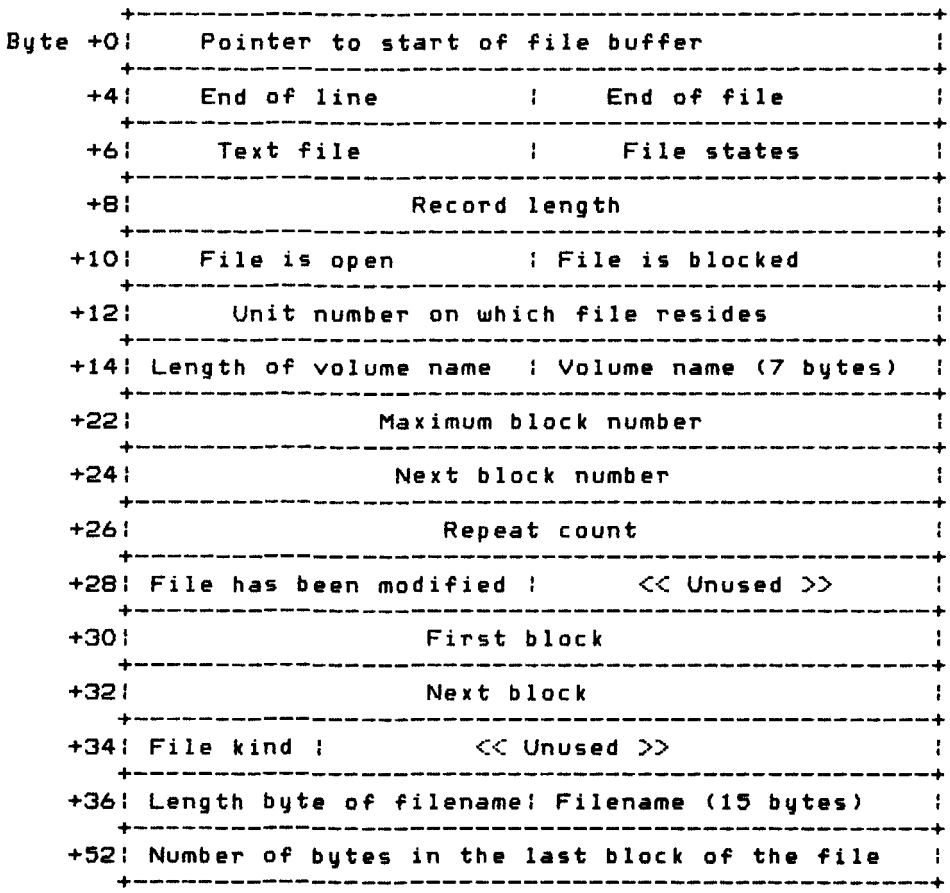
PTab now points to the system device table.

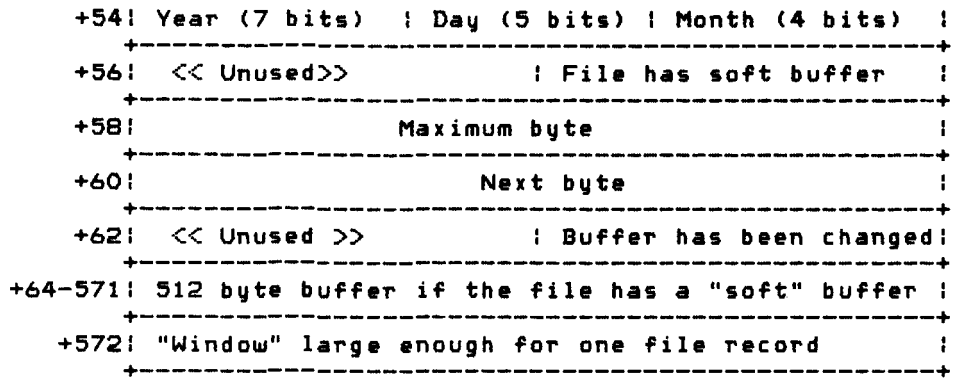
The File Information Block

Each open file has an associated file information block or FIB. A FIB must be created before a file can be opened. The FIB describes the type of file, buffering, etc. to CCOS so that the file can be handled correctly. FINIT will initialize a FIB passed by the user; FOPEN will associate the FIB with a particular file.

The size of the FIB depends upon the type of file being opened. Files accessed with block I/O, such as Pascal untyped files, have 64-byte FIBs, in addition to a user-allocated block buffer. Unblocked files, i.e. TEXT files or typed files, have a FIB that is 576 bytes long, plus enough buffer space to hold one record.

The structure of a FIB is given below:





Pointer to file buffer: A pointer to the buffer at the end of the FIB. Only valid for TEXT or untyped files.

End of line: A Boolean value that is TRUE if an end-of-line character was encountered in the last read of the file.

End of file: A Boolean value that is TRUE if the file is currently positioned at its end.

Text: A Boolean value that is TRUE for TEXT or INTERACTIVE files.

State: A field that can have the values 0-3, and is valid only for text files. Represents the state of the files's buffer and is used by FGET.

Record size: A word quantity denoting the number of bytes in a record.

File is open: A Boolean value that is TRUE if the file is currently open. The fields AFTER this field are valid ONLY if this field is TRUE.

File is blocked: A Boolean that is TRUE if the file resides on a blocked device (e.g. a disk).

Unit number: A word that contains the unit number for the volume the file resides on.

Volume name: A 8 byte string that contains the name of the volume the file resides on. The first byte contains the length of the volume name.

Maximum block: A word denoting the number of the last block in the file.

Next block: A word value containing the number of the next block to be read from or written to the file. Valid only for blocked devices.

Repeat count: A word value representing the number of leading spaces on a line. Included for UCSD file compatibility. Valid only for current record of text files. The normal sequence is DLE N, where N is a binary number representing the number of spaces at the start of the line.

Modified: A Boolean value that is set to TRUE if the contents of the file have been changed.

Header: This is the current directory entry for the file. It contains all fields from byte +30 to byte +56.

Soft buffer: A Boolean value that indicates that the file buffer for this file is part of the FIB, as opposed to being separately allocated as in the case of a blocked file. The following fields are valid only if the soft buffer field is TRUE.

Next byte: A word pointer to the next byte to be read from or written to the current file buffer.

Maximum byte: A word quantity that is the number of the last byte in the buffer.

Buffer changed: A Boolean that is TRUE when the buffer for this file has been changed. Used to notify the system that the buffer must be written to the disk.

Buffer: For typed files only, the 512-byte buffer area at the end of the FIB.

Record window: For typed files, a buffer area large enough to hold one record of the file.

Accessing a FIB from Pascal

The following program fragment illustrates a technique that may be used to access a FIB from Pascal. This example assumes the existance of a file called TEST.

```
Program GetFib;

Uses {$U /ccutil/os.globals.obj} globals;
```

```
Var
  DemoFile : File;
  PtrFib   : PFIB;

Begin
  Reset(DemoFile,'TEST');
  PtrFib := @DemoFile;
```

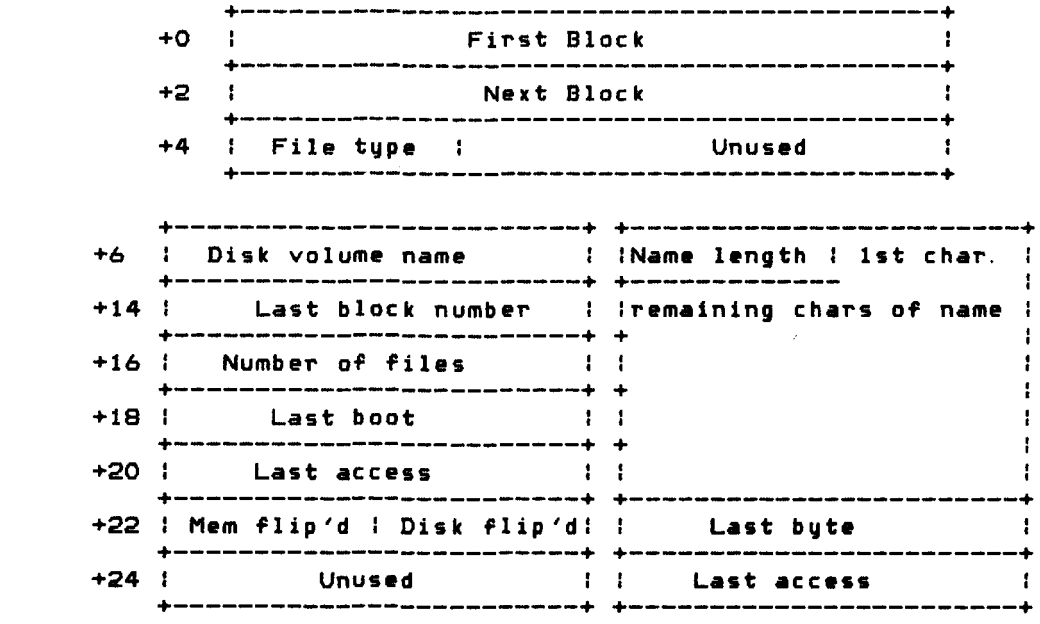
PtrFib now points to the FIB of DemoFile.

The Device Directory

Directories reside on blocked devices (i.e. disk volumes), and contain information about files on the volume, such as the size of the file, its type (text file, code file, etc.), its starting location on the disk, and the date of its last modification.

The volume directory has space for a maximum of 77 entries, so a volume may have a maximum of 77 files, regardless of the amount of space available on the volume. In addition, each directory contains a special entry which describes the particular volume. This header record contains information on the volume in a format similar to that of a directory entry for a file.

The structure of a directory entry is shown below. The first part of the structure, i.e. the first 6 bytes, is common to all directory entries. Of the last portion, the left part represents the structure of the entry if it is a volume header, and the right part represents the structure of the entry if it is a directory entry for a file.



Volume header entry File directory entry

First block: A word quantity denoting the starting block

number of the file. For a volume header, denotes the first available block on the disk, normally 0.

Next block: A word denoting the next available block after the end of the file. For a volume header, the first available block on the volume, normally 6, since the directory and other systems data occupy blocks 0 through 5.

File type: A four-bit quantity which designates the type of file this entry represents. The possible values for this are as follows:

0 or 8 - directory header entry

2 - code file

3 - text file

5 - data file

The "code file" type designation refers to a P-system code file. CCOS code files are flagged as data files. This methodology was adopted for P-system compatibility.

The remaining 12 bits of the field are unused.

If the directory entry is for a file:

Name length: A one byte field containing the number of characters in the file name, up to a maximum of 15.

Name: The actual file name. This field is actually part of a string, with the name length field as the string length specifier.

Last byte: A word containing the number of characters in the last block of the file. The remainder of an end block is padded with ASCII null characters.

Last access: A word containing a date record representing the last time the file was changeced.

If the directory entry is a volume header:

Disk volume name: An 8 byte field consisting of a length byte followed by up to 7 characters of volume name.

Last block: A word denoting the number of the last available block on the volume.

Number of files: A word containing the number of files on the volume (maximum of 77).

Last access: A date record specifying the last write access to the directory.

Last boot: A word containing the most recent setting of the date. This is updated automatically when CCOS is booted. Only the record on the boot volume is affected.

Memory flipped: A Boolean used by the system when a directory read into the Filer is byte flipped.

Disk flipped: A Boolean used by the system to indicate that the disk directory is byte flipped.

Accessing device directories from Pascal

Routines to read and write device directories are contained in the CCdirIO unit in CCLIB. See "The System Library Users Guide" for more information.

IORESULT codes

The IORESULT field in the system communications area (SYS-COM) is an integer value that is set every time an I/O operation is performed. This value is available in Pascal by calling the predeclared function IORESULT, as in:

```
IF IOResult <> 0 Then Begin...
```

The possible values for IORESULT are:

0: Good result. IO operation completed successfully.

1: Bad block. Usually caused by a CRC error when reading a disk.

2: Either a nonexistent unit number has been referenced, or no driver is installed for the unit.

3: An attempt was made to perform an invalid output operation, generally a unit I/O request not supported by the driver in use.

4: Nebulous Hardware Error.

5: Lost device: A previously valid device went offline.

6: Lost file: A prevously accessed file has disappeared from the directory.

7: Invalid file name.

8: No room left on the device (usually caused by an attempt to write a file larger than the remaining space on a disk).

9: Fatal IO error: A device has become unavailable in the middle of an IO operation.

10: No such file- The file specified does not exist.

11: Duplicate file name: Attempt to create a file on a device which already has a file of that name resident.

12: File is already open: Attempt to open a previously opened file.

13: File is not open: Attempted IO operation on a closed or unitialized FIB, such as an attempt to read a file without opening it.

14: Bad format: Non-numeric data was read in an integer or real format READ operation.

15: Ring buffer overflow. Currently unused by system.

16: Write protect error: Attempted write to a write protected device.

17: Seek error: Attempted seek on a file that is not a blocked or TEXT file. Also caused by a seek to a negative record number, or a floppy track seek error.

18: Invalid block number. A request was made to a blocked device with a block number that was negative or greater than the highest block number on the device.

24: Device timeout. The system has timed out while waitting for a device to respond. Currently used only with floppy disks.

25: Attempted seek to track 0 of a floppy failed.

26: Failure to read or write floppy diskette, usually indicates an unformatted disk.

27: Invalid sector length on floppy disk. Usually indicates an attempt was made to read a diskette formatted on another system.

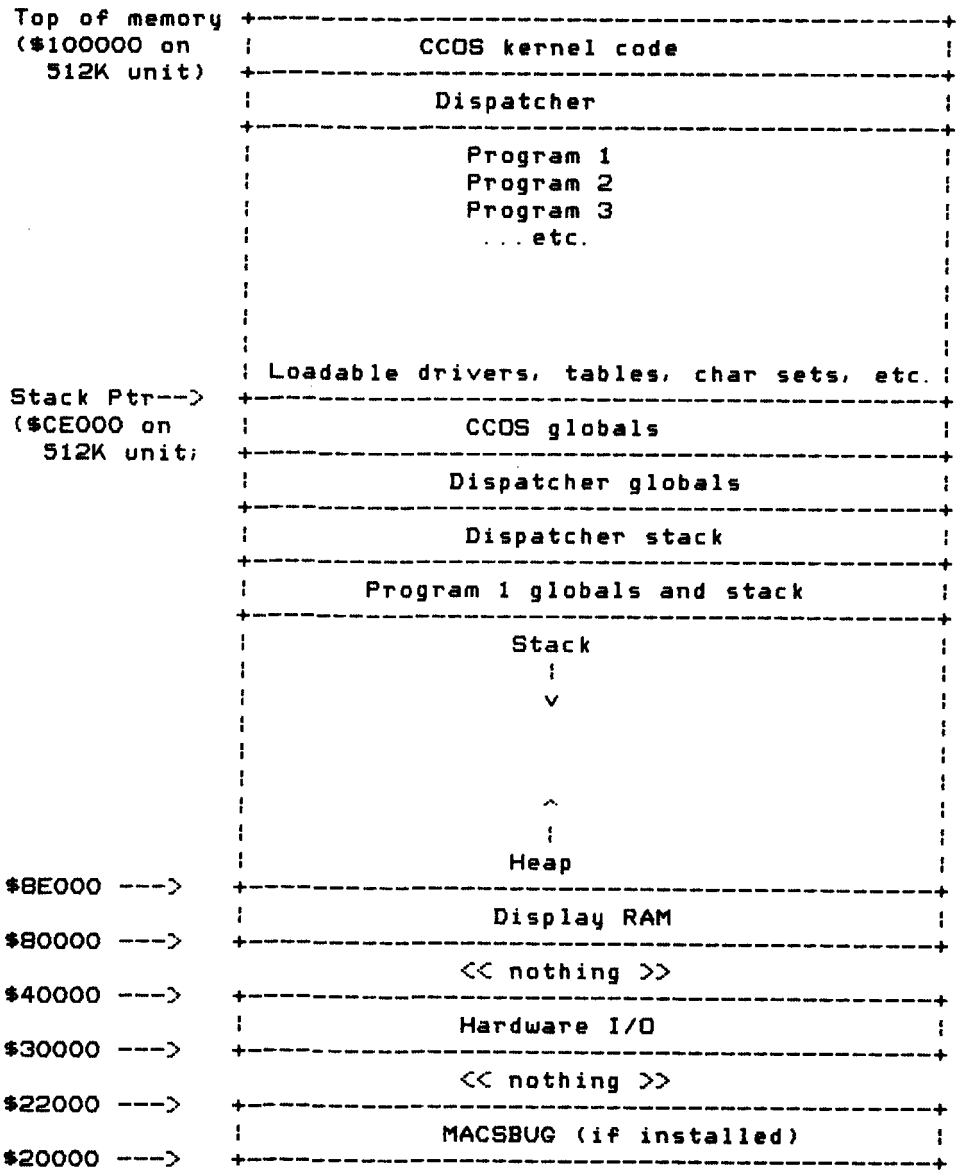
28: Floppy track read was not the same as the track requested. Usually indicates foriegn diskette format or clobbered disk.

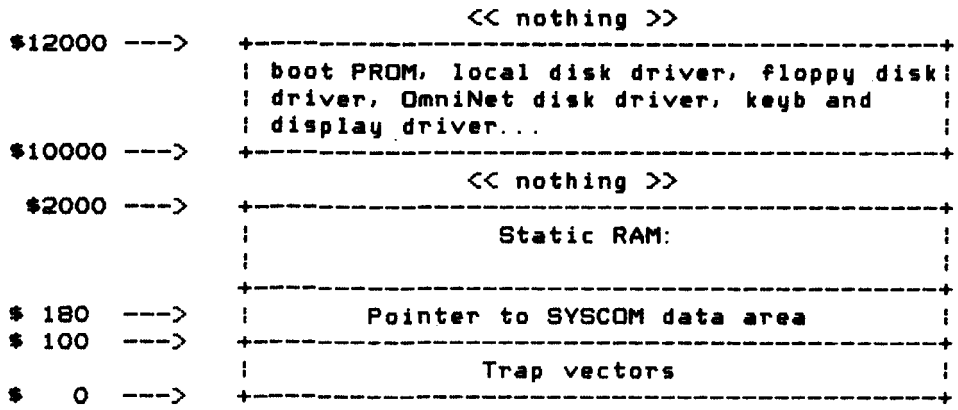
29: Track read was flagged as a bad track. (as per IBM spec)

64: Device error of unknown origin.

Some devices on the system may generate device-specific IORESULT codes. IORESULT codes for Corvus supplied device drivers are specified in chapter 5, "Standard Device Drivers."

CCOS Resource Usage and Concept Memory Map





The system stack pointer determines how memory is apportioned between code space and data space. The current setting of the system stack pointer can be seen by typing:

SP [Return]

at the dispatcher level. The stack pointer value may be changed from the dispatcher by typing:

SP <newvalue> <# of "K" RAM avail>

This causes the system to warm boot and reallocate its memory. The second parameter is optional. For example, a 512K system could be turned into a 256K system by typing:

SP 9C400 256

Users requiring a different allocation of code and data space can force the system to come up with any desired stack pointer setting by putting an "SP <pointer value> STARTUP" line in the startup file STARTUP.TEXT on unit 5. This will lengthen the amount of time the system takes to come up since the system will first perform a normal boot, then immediately warm boot to reset the stack pointer.

The STARTUP at the end of the line must be included, or the system will reboot in an infinite loop.

The system stack pointer should not be confused with the 68000 stack pointer; they are completely seperate.

In a 512K machine with a standard complement of drivers, approximately 170K of code space and 200K of data space is available with the standard stack pointer setting.

Program code is loaded starting at the top of available memory (initially, just under the dispatcher) and grows downwards. CCOS drivers are loaded starting at the stack pointer and grow upwards. The address of the current top of driver space is contained in the pointer starting at $10C, while the current bottom of code space is contained in the pointer starting at $108. The amount of available code/driver space is the difference between these two values.

Static data structures (simple variables, constants, and arrays) are allocated starting at the stack pointer and growing downward towards low memory addresses; this data is collectively referred to as the stack. Dynamic data structures, such as those created with NEW in Pascal, are allocated starting at the top of display RAM and growing upwards; this data structure is called the heap. Data memory is full when the stack and the heap collide; code memory is full when the drivers and program code collide.

The Pascal function MEMAVAIL returns the approximate amount of space remaining between the stack and the heap. The VOLUTIL utility on /CCUTIL can display a graphic and numerical indication of available memory.

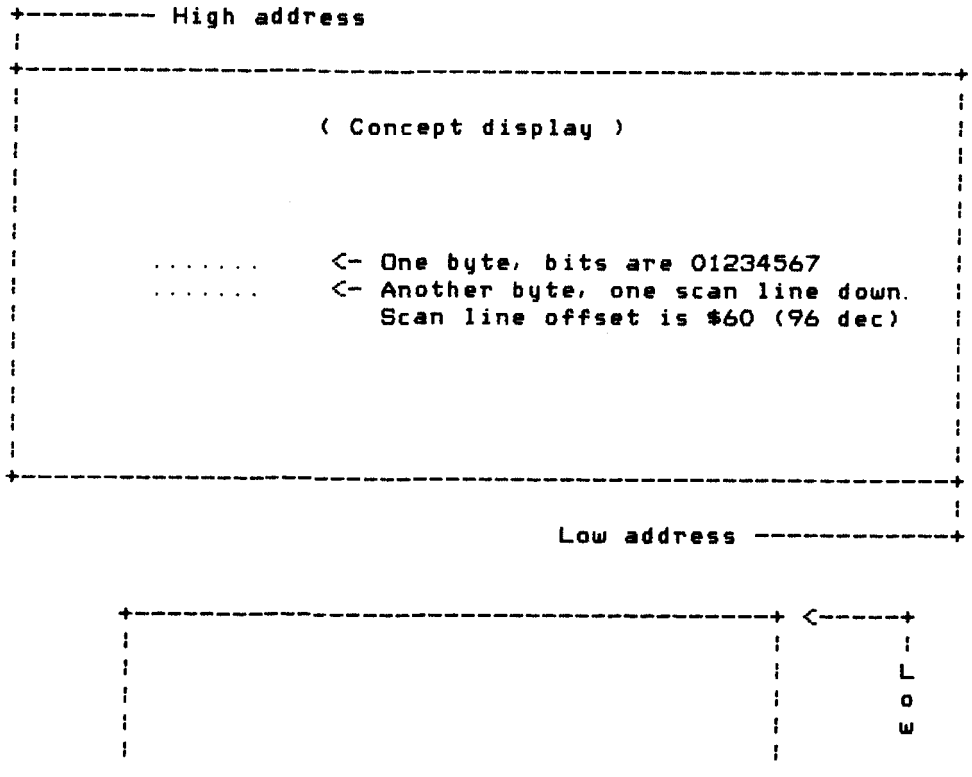
Concept Display Memory

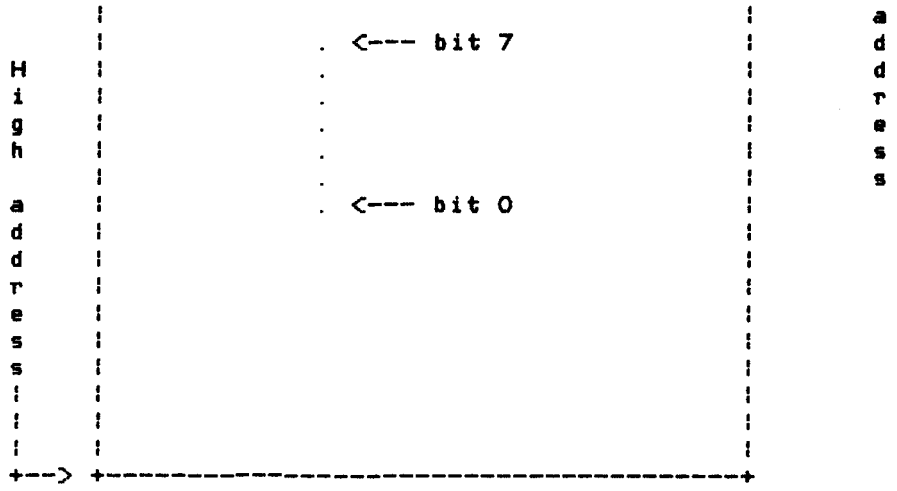
The Concept display is bitmapped-- each pixel on the screen represents the state of one bit of memory. If the bit is a logical 1, the corresponding pixel is light. If the bit is a logical 0, the corresponding pixel is dark.

The display contents are changed by writing values to memory locations between $80000 and $8DFFF (524288 to 581631 decimal).

Screen memory is mapped to the physical display in the same fashion regardless of whether the display is vertical or horizontal. Thus, low memory addresses which reference the lower right hand corner of the display when it is horizontal will refer to the upper right hand corner of the display when it is vertical.

When the screen is oriented horizontally, increasing addresses in memory reference scanlines on the screen from right to left, bottom to top:





The ReadBytes routine in the CCgrfIO library unit can be used to read data directly from screen memory.

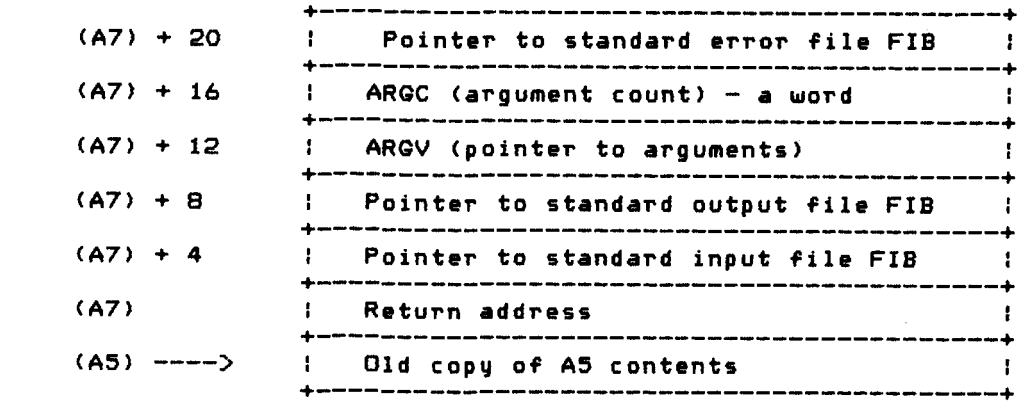
CCOS Register Usage

68000 register usage in CCOS is as follows:

A4- Holds address of overlay jump table
A5- Holds address of user global data
A6- Holds the base address of the local stack frame. A6 contents are undefined for a program at the outermost (main) level.
A7- Holds the current top-of-stack address.

All other register contents are destroyed when system calls are made.

When a program is started, the top portion of the stack contains:



ARGC and ARGV are special values set when a program is invoked from the dispatcher. ARGC points to an integer, while ARGV points to an array of pointers to strings. For example, if you called the linker by typing:

```
Linker [Return]
```

ARGC would be 0 and the ARGV pointer would be invalid. If, however, you invoked it by typing:

```
Linker testprog /ccutil/cclib !paslib [Return]
```

then ARGC would be 3, and ARGV would point to an array of pointers to the memory where the three strings were stored.

Concept I/O Mapping and Interrupt Structure

All Concept hardware I/O occurs in the address range $30000-$3FFFF. This includes:

* The keyboard port;
* The two RS-232C (datacomm) ports;
* The 6522 VIA and real-time clock;
* The OmniNet port;
* The four Apple-compatible 50 pin slots.

Interrupts

Most Concept I/O is interrupt driven. The 68000 user interrupts are not used since the 6502-family devices used for I/O (6522,6551) do not produce vectors. The NMI (non maskable interrupt) is not normally used on the Concept.

Interrupt Levels

Priority Level	Signal Name	Device
7	NMI	Not used
6	NKeyInt	Keyboard
5	NTimInt	VIA timer
4	NSr0Int	RS-232 port 0
3	NOmInt	Omninet
2	NSr1Int	RS-232 port 1
1	NIOcInt	50 pin slots/ datacomm ctrl

Time-critical sections of code can disable interrupts by setting the 68K interrupt priority mask to 7. Since the Concept has no level 7 interrupts, these sections of code can then run without being interrupted.

The following sections detailing CCOS interrupts and the detailed I/O map will necessarily involve some hardware description. For details on the various hardware items identified, consult the "Corvus Concept Hardware Description" manual.

Keyboard Interrupts

The keyboard UART is a 6551 device. It operates in a receive only mode. Each time a new character becomes available, a level 6 interrupt is generated.

Timer Interrupts

The Concept timer is part of a 6522 device. One of the internal counters is used to cause a level 5 interrupt every 50 milliseconds. This is used in repeat key timing, among other things. None of the other interrupt possibilities of the 6522 can be used.

DataComm Interrupts

The Concept's RS-232 ports are serviced by a 6551. This device can be set to interrupt on receiving or transmitting a character. The 6551 can be configured (in software) for a variety of baud rates and data formats (see the 6551 data sheet for details).

The datacomm port 0 (or RS-232 port 0) generates level 4 interrupts. Datacomm port 1 has its own 6551 and is primarily used for driving printers. The only functional difference is that it generates level 2 interrupts.

Omninet Interrupts

Whenever the Concept's internal Omninet transporter completes an operation, it generates a level 3 interrupt. Omninet cannot turn the interrupt off, so NOMOFF must be sent at the end of an Omninet interrupt to turn the interrupt off. Care must be taken not to respond to the same interrupt more than once.

Additional details of Omninet programming may be found in the Omninet Programmer's Guide.

Detailed I/O Map

Note: An "x" in an address means "Don't care". All I/O addresses MUST BE ODD. The 50-pin connectors used in the Concept follows the signal conventions and addressing established in the Apple IIc computer.

This section of the manual is intended for use by programmers familiar with the Concept I/O devices. Detailed technical information on such things as the 6522 and 6551 devices may be found in the "Hardware Description" manual. Detailed information on programming the Omninet hardware may be found in the "Omninet Programmer's Guide."

I/O Ports							
Key board	DComm port1	DComm port2	6522 VIA	Clock ALTMAP	Omni strobe	Omni intrpt	I/O strobe
30F0x	30F2x	30F4x	30F6x 30F7x	30F8x	30FAx	30FCx	39FFF

ROM mapping of I/O slots (Apple addresses in parens)					
Slot #	Byte 0	Byte 1	Byte 2	...	Byte N
1	(C100) 30201	(C101) 30203	(C102) 30205	... ...	(C1FF) 303FF
2	(C200) 30401	(C201) 30403	(C202) 30405	... ...	(C2FF) 305FF
3	(C300) 30601	(C301) 30603	(C302) 30605	... ...	(C3FF) 307FF
4	(C400) 30801	(C401) 30803	(C402) 30805	... ...	(C4FF) 309FF

Note: The initial $Cxxx of an Apple address is replaced by $30xxx. The lower three nibbles of the Apple address are shifted left one bit and 1 added. ROM tables and ID may be read and used, but the 6502 code cannot be executed. Device drivers in 68000 code may be written and linked to devices with the ASSIGN utility.

I/O slot register addresses				
I/O register	Slot 1	Slot 2	Slot 3	Slot 4
0	30021	30041	30061	30081
1	30023	30043	30063	30083
2	30025	30045	30065	30085
3	30027	30047	30067	30087
4	30029	30049	30069	30089
5	3002B	3004D	3006B	3008B
6	3002D	3004B	3006D	3008D
7	3002F	3004F	3006F	3008F
8	30031	30051	30071	30091
9	30033	30053	30073	30093
A	30035	30055	30075	30095
B	30037	30057	30077	30097
C	30039	30059	30079	30099
D	3003B	3005B	3007B	3009B
E	3003D	3005D	3007D	3009D
F	3003F	3005F	3007F	3009F

Note: These addresses correspond to the Apple device control addresses, which are of the form $C0x0 - $C0xF, where x is the slot number + 8. Unlike the Apple, these addresses are NOT CONTIGUOUS, but alternate bytes. This is an artifact of the 68000 16 bit data bus structure.

6522 VIA general purpose I/O				
30F61	Output register B, input register B			
	0	Video off		Output
	1	Video address 17		Output
	2	Video address 18		Output
	3	Horizontal/vertical switch		Input
	4	CH rate select	DC0	Output
	5	CH rate select	DC1	Output
	6	Boot switch 0		Input
	7	Boot switch 1		Input
30F63	Output register A, Input register A handshake			

30F65	Data direction register B-Set to 37 by boot PROM
30F67	Data direction register A-Set to 80 by boot PROM
30F69	Timer 1 latch low byte, write latch, read cnter
30F6B	Timer 1 latch high byte
30F6D	Timer 1 latch low byte
30F6F	Timer 1 latch high byte
30F71	Timer 2 latch low byte, write latch, read cnter
30F73	Timer 2 counter high byte
30F75	Shift register
30F77	Auxillary control register
30F79	Peripheral control register
30F7B	Interrupt control register
30F7D	Interrupt enable register
30F7F	Output register A, input register A, no handshake

0	Omninet ready	Input
1	Clear to send (Dtacom 0)	Input
2	Clear to send (Dtacom 1)	Input
3	Data set ready (Dtacom 0)	Input
4	Data set ready (Dtacom 1)	Input
5	Data carrier detect (Dtacom 0)	Input
6	Data carrier detect (Dtacom 1)	Input
7	Exclusive OR of above signals for interrupt	Output

30FA1 - 30FBF	Omninet transporter resgister

30FC1 - 30FDF	Reset Omninet interrupt

Register	Keyboard	Data Comm 0	Data Comm 1
Data	30F01	30F21	30F41
Status	30F03	30F23	30F43
Command	30F05	30F25	30F45
Control	30F07	30F27	30F47

System Calls | 3

This chapter provides a complete description of the system call interfaces. System calls are made to CCOS by pushing parameters on the 68000 stack, and performing a JSR to the desired routine. Results from the call, if any, are returned on the stack. Unit and file I/O calls will also set the value of IORESULT, the first field in SYSCOM.

Since CCOS uses many of the 68000's internal registers, the SAVEM.L instruction should be used to save the register contents prior to the actual call of the system routine.

Parameters for each call are described in the order in which they should be pushed. All system calls act as either Pascal proceddres or Pascal functions. The system calls which are procedures return the stack as they received it, with various fields changed- i.e. the stack pointer will be in the same place, and the user must remove the various items from the stack before proceeding. System calls which act as Pascal functions will return a single value on the stack, the type of value returned depending upon the function definition. Space for the result must be pushed onto the stack before any of the parameters the function may take.

One-byte values (such as Booleans) are pushed onto the stack with the MOVE.B instruction. This actually pushes a word onto the stack and decrements the stack pointer by two. The low address of the word (or the most significant byte of the word is considered as an integer) contains the actual one byte value desired.

The entry points to the various system routines are kept in the system call vector table. The location of the table may be found in SYSCOM. SYSCOM and the system call vector are described in chapter 1. The various data structures referenced by these calls (such as directories, file entries, etc.) may be found in chapter 6: "OS Global Declarations."

The discussions below cover the following topics:

* Unit I/O
* File I/O

* Memory management

Warning: Direct system calls should be used with a great deal of caution! Conflicts can arise when direct system calls are made since the supporting action of the high level language is bypassed.

Unit Input/Output

Unit I/O is the lowest level of the CCOS I/O facilities. Unit I/O communicates with system devices in terms of BLOCKS (chunks of 512 bytes) on blocked devices and characters on character devices. All unit I/O procedure and functions set the syscom field IORESULT. Programs using unit I/O should check this field after every operation.

Unit I/O is accessed by pushing a group of parameters on the stack, and then performing a call to the specific procedure desired.

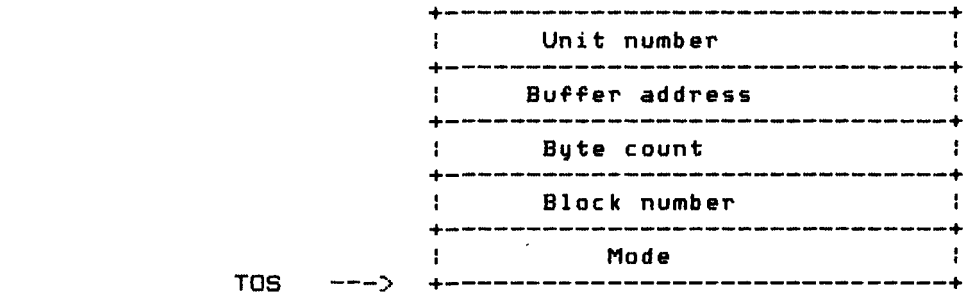
Five system interfaces are provided for unit I/O:

* UNITREAD- Read a unit;
* UNITWRITE- Write to a unit;
* UNITBUSY- Check to see if unit is in use;
* UNITCLEAR- Reset unit;
* UNITSTATUS- Device dependent functions.

These system interfaces are described in detail in the following sections.

UNITREAD and UNITWRITE

UNITREAD and UNITWRITE are procedures which transfer data between a memory buffer and a specific unit. Parameters for these calls are:



Unit number- A word quantity representing the unit number involved in the transfer.

Buffer address- A pointer to the memory buffer.

Byte count- A word quantity containing the number of bytes to be transferred.

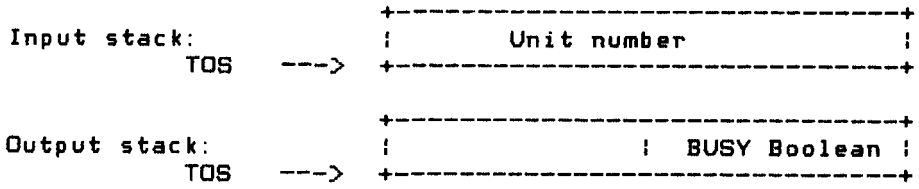
Block number- A word quantity containing the number of the starting block to be read or written. This parameter is ignored by character units such as printers or the keyboard.

Mode- A word quantity that affects device-dependent characteristics. The characteristics affected will depend on the device driver and are documented for each driver.

UNITBUSY

UNITBUSY is a function which is called to see if a unit is ready for I/O. For input purposes, this generally means that the unit queried should have characters ready. For output, it simply means that the unit is ready to accept data.

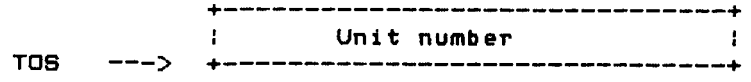
UNITBUSY takes only one parameter, a word quantity containing the number of the unit desired. The result returned is a Boolean which is TRUE if the unit is busy, or FALSE if it is not. The UNITBUSY function will overwrite the top of stack with the result. Note that this is different from the other UNIT intrinsics.



Only a single byte is removed from the stack to get the UNITBUSY result. This is accomplished with a MOVE.B (SP)+,Dn instruction where Dn is a data register from D0 through D7.

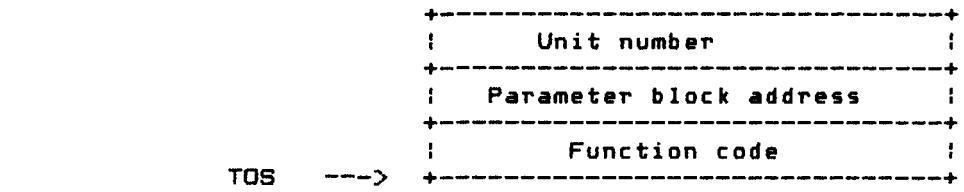
UNITCLEAR

UNITCLEAR is a procedure used to reset a unit to a known state. Like UNITBUSY, it takes a word parameter containing the number of the unit. The exact function of UNITCLEAR is device dependent, but in general it serves to cancel any pending I/O to the unit and to flush any associated buffers.



UNITSTATUS

UNITSTATUS is a catch-all procedure designed to return the current status of a unit, as well as to change various unit parameters. The parameters affected depend upon the unit driver; see the appropriate driver documentation for details.



Unit number- A word quantity representing the unit number.

Parameter block-A long word pointer to the buffer for the data passed to and from the unit.

Function code- A word quantity the selects the function to perform.

UNITSTATUS may or may not return results in the parameter block, depending upon the operation performed.

File Input/Output

This section describes the CCOS facilities for direct file I/O. Before file I/O can be performed, a File Information Block (FIB) must be allocated (see chapter 2 for details of FIB structure). In addition, blocked files require a buffer of sufficient size to accomodate the largest amount of data to be transferred at any one time.

From the programmer's point of view, there are three types of data files: typed files, text files, and untyped files. A typed file is a file whose records follow some sort of structure, and are declared FILE OF <whatever>;. The GET, PUT, and FSEEK command in Pascal are used to manipulate these files.

CCOS text files exmploy the UCSD text file structure. Leading blanks on a line are compressed to the form DLE N, where DLE is the ASCII Data Link Escape code (decimal value 16), and N is a binary number representing the number of spaces compressed. Thus, the numbers 16 8 at the beginning of a line indicate that 8 spaces exist at the start of the line. In addition, lines in text files may not be broken across 1024-byte "pages." The first two blocks in a text file comprise the first page. If a line will not fit on the current page, it is moved to the next page, and the remaining bytes of the current page are set to ASCII Null characters (decimal 0). From Pascal, the READ, READLN, WRITE, and WRITELN procedures are used to access text files. The DLE-blank compression and page structure are handled automatically.

Untyped files are declared as simply FILE;, and are treated as collections of blocks with no other structure. The BLOCKREAD and BLOCKWRITE functions are used to access these files from Pascal.

The basic file I/O procedures are FINIT, FOPEN, FPUT, FGET, FCLOSE, READCHAR, and WRITECHAR.

All strings pointed to (such as volume, device, and path names) are Pascal strings: the first byte contains the number of characters in the string.

FINIT

FINIT is a procedure that creates a File Information Block (FIB). While FINIT is handled automatically in Pascal programs, machine code programs must call it to establish a FIB prior to calling FOPEN.

volume name is used, the default volume is assumed. The boot volume may be abbreviated as !, i.e. !/<filename>.

Pointer to FIB- A pointer to the file information block.

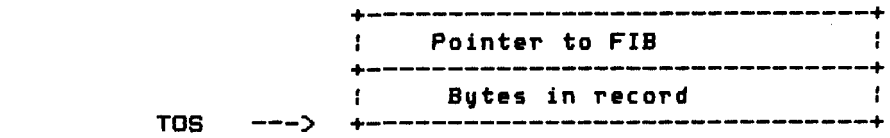
New file indicator- A Boolean which, when set to TRUE, indicates that a new file is being created.

FGET and FPUT

The FGET and FPUT procedures operate identically with the exception of the direction in which the data is transferred. The only parameter to either procedure is:

Pointer to FIB- A pointer to the file information block.

Both FGET and FPUT transfer one record of the current file to or from the buffer pointed to by the FIB. The SEEK procedure can be used to position the file at the desired record prior to using FGET or FPUT. FGET and FPUT cannot be used with untyped files.



Pointer to FIB- A long word pointer to the file information block.

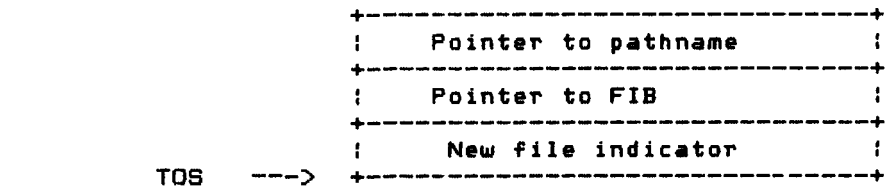
Bytes in record- A word quantity which, if positive, represents the number of bytes in each record of the file. If zero, the file is an interactive file, such as the keyboard, and is handled in a manner similar to text files, with some minor differences in the treatment of end-of-line. There are two possible negative numbers:

-1: File is a UCSD P-system compatible file. These are untyped files (i.e. VAR UCSDFile : File;), and the user must provide the file buffer. Only block I/O may be performed.

-2: File is an ISO Standard Pascal compatible file. These are text files, i.e. VAR ISOFile : Text;.

FOPEN

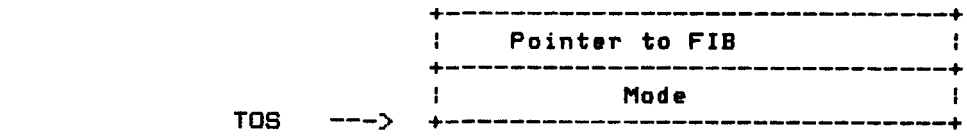
The FOPEN procedure opens a file for data transfer. The parameters are:



Pointer to pathname- A pointer to a character string containing the pathname of the file to be opened. Currently, a maximum of 24 characters may be in this string: a 7 character volume name enclosed with "/" slash characters, followed by a file name of up to 15 characters. If no

FCLOSE

The FCLOSE procedure closes a file, first flusing any I/O buffers associated with the file. The actual file is "disposed" of in a manner determined by the MODE parameter. The parameters for FCLOSE are:



Pointer to FIB- The pointer to the file information block.

Mode- A word quantity indicating the disposition of the file after it is closed. Possible values are:

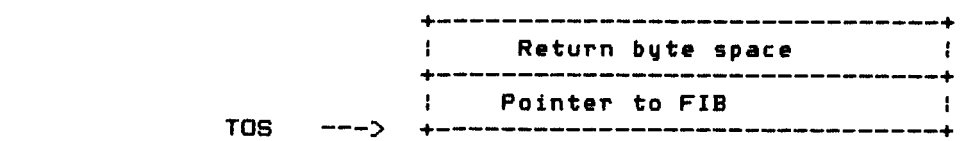
0: NORMAL- If the file existed prior to the FOPEN condition closed by this FCLOSE, it is retained. If it was created by the FOPEN, it is purged from the file system.

1: LOCK- Just like NORMAL, except the file is ALWAYS retained, even if it was created by the the FOPEN.

2: PURGE- The file is always removed from the file system, even if it previously existed.

READCHAR

The READCHAR function reads a single character from a file. It can only be used with file of type INTERACTIVE (mode 0) or TEXT (mode -2). The parameters for READCHAR are:



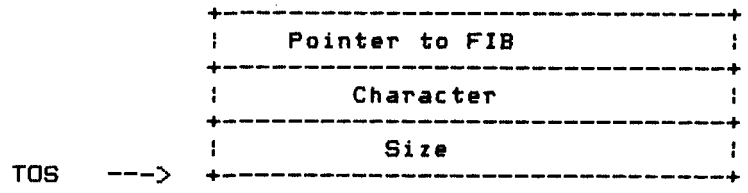
Return byte- The user must push a byte quantity on the stack prior to executing this call. The byte read from the file will be returned here.

Pointer to FIB- A pointer to the file information block.

READCHAR returns a single character on the top of the stack.

WRITECHAR

The WRITECHAR procedure writes a character to a file. An optional field width parameter can be used to cause space filling. Like READCHAR, WRITECHAR can be used only with INTERACTIVE or TEXT files. The parameters are:



Pointer to FIB- A pointer to the file information block.

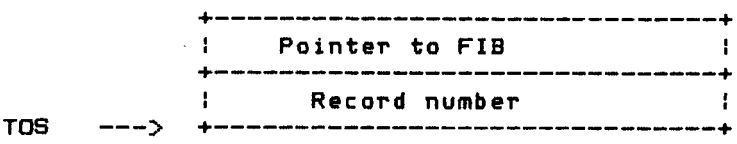
Character- The character to be written to the file.

Size- A word quantity representing a field width, normally set to one. If greater than one, the character written is preceeded with (Size-1) spaces.

SEEK

The SEEK procedure positions the file to the start of a given record. Normally used with typed files (i.e. FILE OF <whatever>) , it can also be used with TEXT files, which are treated as files of character records. Thus, seeking to position 120 of a TEXT file would leave the file pointer positioned at character 121 (record numbers start with 0).

Parameters to SEEK are as follows:



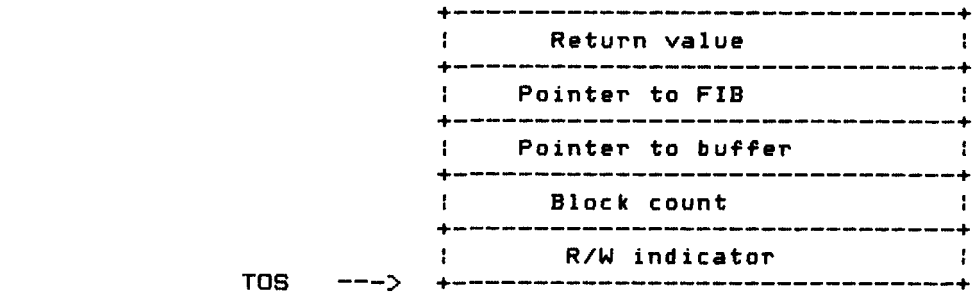
Pointer to FIB- A pointer to the File Information Block.

Record number- A long word quantity representing an

absolute record number.

BLOCK I/O

Block oriented file I/O is used to read and write entire blocks (groups of 512 bytes) from disk files. The Block I/O procedure can only be used with untyped files- files created with a mode of -1. Parameters to BLOCKIO are as follows:



Return value- An integer whose value is set by BLOCKIO.

Pointer to FIB- A pointer to the File Information Block.

Pointer to buffer- A pointer to the area of memory to be used for the data transfer.

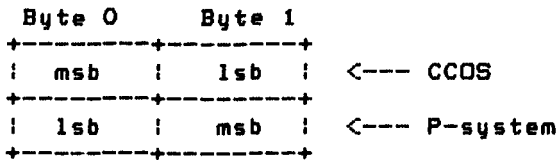
Block count- A word quantity representing the number of blocks to be transferred.

R/W Indicator- A Boolean quantity which indicates a read when TRUE and a write when FALSE.

BLOCKIO returns a word quantity on top of the stack. If this value is non-zero, it represents the number of blocks actually transferred. It is important to note that this may not be the same as the number of blocks requested- as might occur when an end-of-file condition is encountered. If the value is zero, some form of error has occured.

FlipDir

The FlipDir procedure "flips" a directory- i.e. it changes the order of bytes in integers from high, low to low, high in certain fields in the directory. FlipDir is allows CCOS to access P-system directories, which are flipped with respect to CCOS directories.

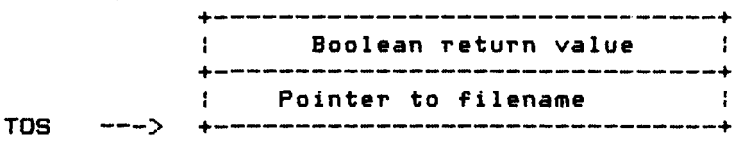


Specifically, FlipDir flips all the integer fields- words 0, 1, 3, 7, 8, 10, 11, and 12 within a given directory. The fields affected are FirstBlock, LastBlock, Misc, DeovBlock, DNumFiles, DLastBoot, DLastByte, and DAccess. FlipDir takes as parameter the address of a directory in memory- a directory must be read from disk into a memory buffer before FlipDir can be used.

Pointer to directory- A word pointer to a directory data structure in memory.

ValidName

The ValidName function returns a Boolean TRUE on the stack if the file name passed to it is valid. Validity checking is simple: if the name is of the right length and contains only characters in the set ['A'..'Z','0'..'9',' ','-','_'], then it's valid.



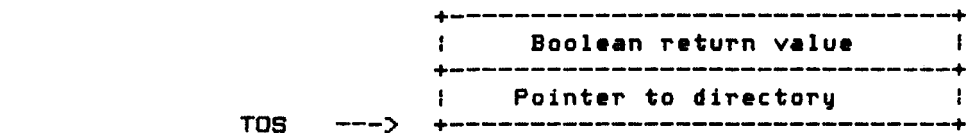
Boolean return value- One word space for return value.

Pointer to file name- A pointer to a file name string in memory.

ValidDirectory

This function returns a Boolean TRUE on the stack if the directory passed is valid. The directory to be checked must

first be read into memory with GetDir.

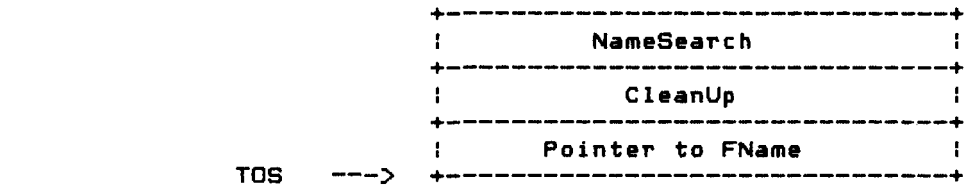


Boolean return value- A word space for the return value.

Pointer to directory- A pointer to a directory data structure.

GetVolNames

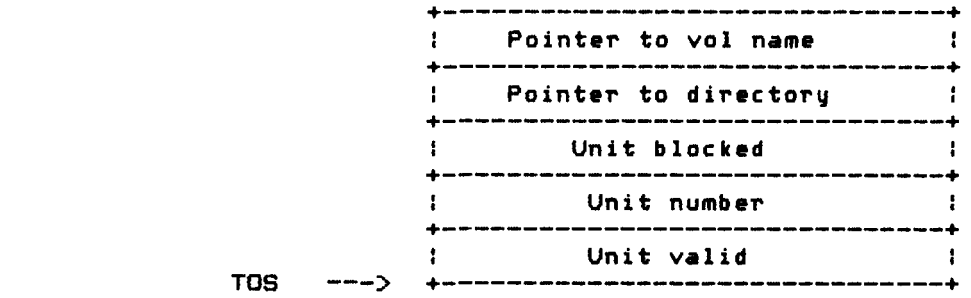
Presumably gets vol names, but I don't really know...



I also don't know what any of these parameters are...

GetDir

This procedure is used to read a directory into memory from a blocked device.



Pointer to vol name- A pointer to a volume name.

Pointer to directory- A pointer to a directory.

Unit blocked- A pointer to a Boolean value which is TRUE if the device is blocked.

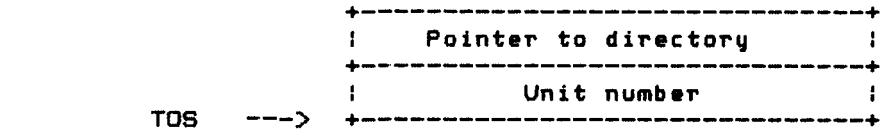
Unit number- A pointer to a word containing the device number.

Unit valid- A pointer to a Boolean which is TRUE if the device is valid.

GetDir will attempt to read a directory from the given device. If the device indicated is not a blocked device or has an invalid directory, GetDir will start searching all volumes, starting with the next highest device number, and continue until all volumes have been searched or a valid directory is found. Thus, the user should always check the device number on the stack when the routine returns to make sure that the directory returned was from the requested volume.

PutDir

The PutDir procedure writes a directory to a given unit.

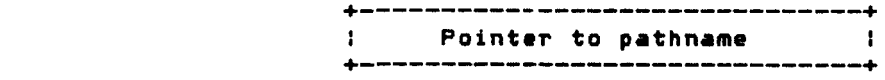


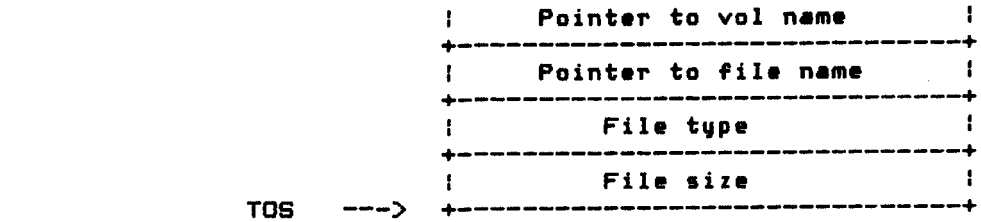
Pointer to directory- A pointer to a directory structure in memory.

Unit number- A word value containing the unit number of the device.

CrackPathName

The CrackPathName procedure takes a pathname (such as "/CC-SYS/ED"), determines if it's valid, and returns seperate volume and file names, as well as the file type and size of the file, in blocks.





Pointer to pathname- A pointer to the pathname.

Pointer to vol name- A pointer to the volume name returned by the procedure.

Pointer to file name- A pointer to the file name returned by the procedure.

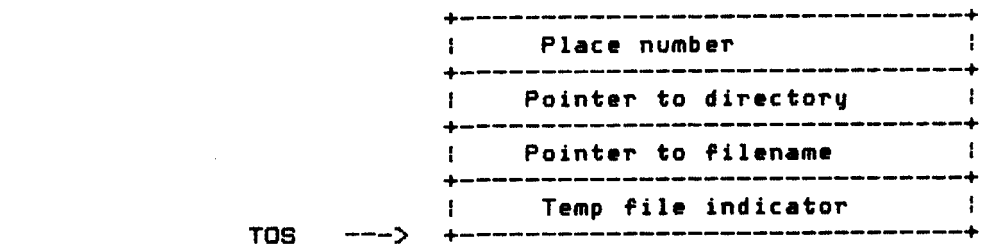
File type- A word quantity designating the file type. Legitimate values are:

0: Directory	5: Data file
1: XDSKFile	6: Graffile
2: Code file (P-system)	7: Foto file
3: Text file	8: Securdir
4: Info file	

File size- A word quantity containing the size of the file, in blocks.

SearchDir

The SearchDir procedure searches a given directory for an occurence of a given file name. If the name exists within the directory, its place number (i.e. 1 for first file, 2 for second file, etc.) is returned. 0 is returned if the file is not found.



Place number- A word for the returned value.

Pointer to directory- A pointer to a directory structure.

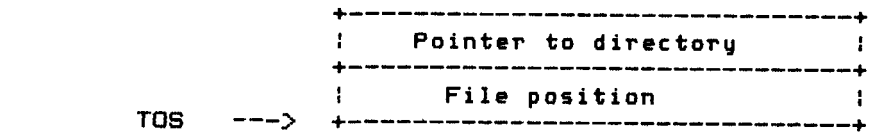
Pointer to file name- A pointer to the name to be searched for.

Temp file indicator- A Boolean which, if TRUE, causes the search to occur only on existing temporary file. If FALSE, onlt non-temporary files are searched.

Note: No IORESULT error is possible with this procedure.

DelEntry

The DelEntry procedure removes a file's directory entry.



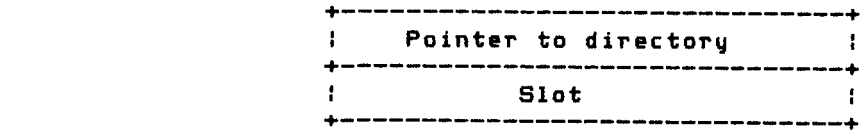
Pointer to directory- A pointer to a directory.

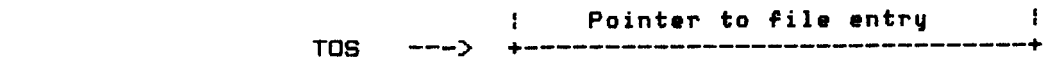
File position- An integer indicating a file's position in the directory, from 1 to 77.

Note that to use this procedure, a directory must first be read into memory with GetDir, and the file's position located with SearchDir. No IORESULT error is possible with this procedure.

InsertEntry

The InsertEntry procedure inserts a file entry into a directory. No checking on the validity of the entry is performed; it is the user's responsibility to make sure that the entry is well-formed.





Pointer to directory- A pointer to the directory.

Slot- A word quantity denoting the position in the directory the new entry will occupy (1-77).

Pointer to file entry- A pointer to the file entry.

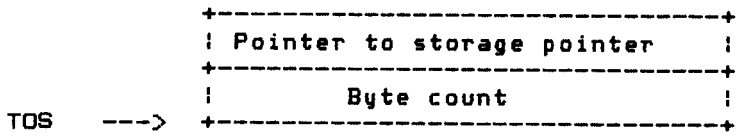
Note: No IORESULT error is possible with this procedure.

Memory Management

This section describes CCOS calls dealing with memory management. The memory allocation is performed on the heap, a dynamic data structure that grows upwards from the bottom of data memory. The heap is used for all dynamic memory work. The stack, on the other hand, grows downward from the top of data memory, and is used for local variables. When the stack and the heap collide, the system dies a messy death.

NEW

The NEW procedure allocates storage on the heap. The parameters to NEW are:



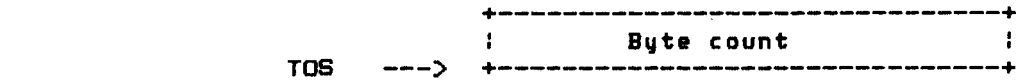
Pointer to storage- A pointer which points to another pointer. The second pointer receives the starting address of the newly allocated storage, assuming that there is enough heap space to process the call. NEW always returns a pointer that is assigned to a word boundary.

If the second pointer is NIL (0), then there is insufficient memory for the data structure.

Byte count- A word quantity representing the number of bytes to be allocated. NEW will round an odd byte count up to the next even number and allocate that number of bytes.

DISPOSE

Currently, DISPOSE is not implemented and acts as a No-Op. It returns a NIL pointer to the caller, and no storage space is deallocated. The parameters to DISPOSE are as follows:



Pointer to storage- A pointer which points to another long word pointer which points to the address of the storage space to be deallocated.

Byte count- A word quantity representing the number of bytes to be freed.

MARK and RELEASE

The MARK and RELEASE procedures are used together to deallocate previously allocated storage. They take the (functional) place of the DISPOSE statement. Both MARK and RELEASE take the same parameters:

Pointer to storage- A pointer which points to another pointer which points to the address of the storage to be marked or released.

MARK is used to "remember" the current position of the top of the heap. RELEASE subsequently uses the pointer returned by MARK to cut the top of the heap back to the previously MARKed position.

MEMAVAIL

The MEMAVAIL function returns a long word quantity which is the number of free bytes between the stack and the heap. This can serve as a rough indicator of the amount of data memory available to the system.

MEMAVAIL takes no parameters.

System Procedure Declarations in Pascal

Following is a summary of the major system I/O routine declarations in Pascal. The parameters are pushed onto the stack from left to right; i.e. for FInit, f is pushed onto the stack, followed by RecBytes. For functions, space for the return values must be pushed onto the stack before the parameters.

All pointer parameters (pfib, pstring64, pbytes) are 4 byte quantities containing the address of the appropriate structure. Integers are two bytes long; long integers are four bytes long. All strings have a preceeding length byte whose contents indicate the length of the string in characters (one byte per character), not including the length byte.

```
Procedure FInit(f: pfib; RecBytes : integer);

Procedure FGet(f: pfib);

Procedure FPut(f: pfib);

Procedure FClose(f: pfib; var mode: Integer);

Procedure FOpen(var fpathname: pstring64;
                var f: pfib;
                Newflag: Boolean);

Function FReadChar(f: pfib): byte;

Procedure FWriteChar(f: pfib; ch: byte; fsize: integer);

Procedure FSeek(f: pfib; frecno: longint);

Function BlockIO(f: pfib; fbuff: pbytes;
                 fblocks, fbock: integer;
                 ReadFlag: Boolean): Integer;

Procedure FlipDir(Var FDir: Directory);

Function ValidName(FName: PString64): Boolean;

Function ValidDirectory(Var FDir: Directory): Boolean;

Function SearchDir(Var FDir: Directory; FTid: Tid;
                   TempFile: Boolean): Integer;

Procedure DelEntry(Var FDir: Directory; Slot: Integer);

Procedure InsertEntry(Var FDir: Directory; Slot: Integer;
```

```
                        Var FEntry: DirEntry);

Procedure GetVolNames(NameSearch, CleanUp: Boolean;
                      FName: PString64);

Procedure GetDir(FVid: Vid; Var FDir: Directory;
                 Var DevBlocked: Boolean;
                 Var FDevno: Integer;
                 Var DevValid: Boolean);

Procedure PutDir(Var FDir: Directory; FDevNo: Integer);

Procedure CrackPathName(FPathName: PString64; Var FVid:Vid;
                        Var FTid: Tid; Var FFKind: FileKind;
                        Var FSize : Integer);
```

Writing A Unit Driver | 4

This chapter discusses the basic structure of a unit driver for CCOS, and presents an example of a driver written in 68000 assembly language.

Calling Conventions

Parameters to a unit driver are passed in the 68000 registers as follows:

D0.W- Unit Number. UNITBUSY returns its Boolean result here.

D1.L- Address of buffer to or from which the data transfer is to be made, or the address of a parameter block.

D2.W- Number of bytes of data to be transferred, or a function code for UNITSTATUS.

D3.W- Block number at which the transfer is to start. (This is applicable only to blocked devices.)

D4.W- Command- determines the operation (unitread, unitbusy, and so on) that the driver is to perform. This parameter is described in more detail below.

D5.W- Mode- a device dependent control whose function is defined in the driver of the device being addressed. It is included to allow control of device operations that are not defined in the standard I/O operations provided by CCOS.

D7.W- The driver passes a completion code (0 indicates successful completion) back to the caller in this register. This becomes the Pascal IORESULT.

Unit Driver Command Parameters

The command passed in register D4.W describes the operation to be performed. The command values are summarized here and described in greater detail below. When a driver receives control from the operating system, the caller has already verified from the unit tables that the given command is valid for the particular driver. The possible values for the command parameter are:

0	UnitInstall -- Install the driver When the operating system installs a unit, either at boot time or when a unit is explicitly assigned, the driver is called with the install parameter. This section performs any initialization code necessary to set up cyclic buffers, place interrupt vectors and so on.
1	UnitRead -- Read from the unit Self-explanatory.
2	UnitWrite -- Write to the unit Self-explanatory.
3	UnitClear -- Clear the unit Reset the device to its initial state. Initialize the device, clear pending interrupts and such.
4	UnitBusy -- Test if unit is busy Check if the unit is ready for data transfer. Driver returns D0.B = 1 (TRUE) if data is ready for transfer, D0.B = 0 (FALSE) otherwise.
5	UnitStatus -- Device dependent control and status. This command is device dependent. Using the function code (D2.W), the driver can return device dependent information to the caller. The buffer address may be used as a pointer to a UnitStatus parameter block.
6	UnitUnmount -- Unmount the unit This command is used when the unit is deassigned. At this time the driver must perform any clean up which includes the restoration of any interrupt vectors the driver might have replaced.

Driver Notes

A driver must be completely relocatable. In addition, it cannot use the stack or heap for storage between calls.

The operating system uses registers A4-A6. Thus, a driver must either not use these registers, or save and restore their contents.

Each I/O slot is assigned a 256 byte area in static RAM. The RAM designated for each slot may be used in any manner by the device in the slot. Additionally, a 512 byte static RAM buffer is available for very temporary operations. This buffer may only be used during a single call to the driver. The static RAM locations are:

```
0900-09FF: Static RAM for slot 1 device
0A00-0AFF: Static RAM for slot 2 device
0B00-0BFF: Static RAM for slot 3 device
0C00-0CFF: Static RAM for slot 4 device

0D00-0EFF: 512 bytes temporary RAM
```

The Concept PROM contains default interrupt handlers. If a driver uses system interrupts, the interrupt vector used by the driver must be restored when the driver is unmounted. The PROM also contains a table of default interrupt vectors which must be used when restoring an interrupt vector during unmount. The PROM locations for the default interrupt vectors are:

10070-10073	CPivec1	level 1 interrupt vector (SLOTS)
10074-10077	CPivec2	level 2 interrupt vector (DC1)
10078-1007B	CPivec3	level 3 interrupt vector (OMNINET)
1007C-1007F	CPivec4	level 4 interrupt vector (DC0)
10080-10083	CPivec5	level 5 interrupt vector (TIMER)
10084-10087	CPivec6	level 6 interrupt vector (KYBD)
10088-1008B	CPivec7	level 7 interrupt vector

A sample unit driver

The following code is the source to the non-interrupt driven datacomm driver. It provides a good overview of driver construction methodologies and architecture.

Note the use of the jump table the driver uses to call its own internal procedures. The use of this jump table makes the driver relocatable since it does not have to be assigned to any one memory area. Remember that all drivers must be relocatable.

```
; File: drv.dcomm.text
; Date: 11-Jan-83

        IDENT   DRVDTACOM
        GLOBAL  DRVDTACOM

;
; DRVDTACOM - The DataComm unit driver
;
; Parameters:   D0.W - Unit number
;               D1.L - Address of buffer
;               D2.W - Count
;               D3.W - Block Number
;               D4.W - Command
;               D5.W - Access Mode
;
;                    Input Parameters:                Result values:
;     Command        Unit  Addr  Count Block Mode     IORESULT  Busy
;
;    0 - Install     D0.W                             D7.W
;    1 - Read        D0.W  D1.L  D2.W  D3.W  D5.W     D7.W
;    2 - Write       D0.W  D1.L  D2.W  D3.W  D5.W     D7.W
;    3 - Clear       D0.W                             D7.W
;    4 - Busy        D0.W                             D7.W      D0.B
;    5 - Status      D0.W  D1.L  D2.W                 D7.W
;    6 - Unmount     D0.W                             D7.W
;

;
;       Some UART equates
;
DcmPort equ     $30f21          ; Slot 6 (datacomm 1) UART pointer
                                ;
Uda     equ     0               ; UART data port offset
Ust     equ     2               ; UART status port offset
Ucm     equ     4               ; UART command port offset
Uct     equ     6               ; UART control port offset
                                ;
RdBit   equ     3               ; Busy bit for input
```

```
WrBit   equ     4               ; Busy bit for output
LfBit   equ     2               ; Auto line feed suppress flag
                                ;
lf      equ     $0A             ; Line feed
cr      equ     $0D             ; Carriage return
cq      equ     $11             ; Control-Q - Start output
cs      equ     $13             ; Control-S - Stop output

; ----  include '/ccos/os.gbl.asm.text'  (not listed)
        list 0
        include '/ccos/os.gbl.asm.text'
        list 1
        page

DRVDTACOM                       ;This is the driver header
        bra.s   comIOrq         ;start of code
        data.b  0               ;device not blocked
        data.b  15              ;valid commands
        data.b  83,01,11        ;date
        data.b  0               ;fill
        data.b  hmlen           ;header message length
hm      data.b  'Non-interrupt DataComm driver'
hmlen   equ     %-hm            ;
                                ;
comIOrq movem.l d1-d6/a0-a6,-(sp) ;Save those registers!
        moveq   #2,d7           ; Set IORESULT to 2
        move.l  d1,a0           ; Data address to A0
        tst.w   d0              ; Is unit number valid?
        blt.s   DcmRtrn         ; No, return (IOresult = 2)
        move.l  pSysCom.W,a2    ; (A2) = SYSCOM
        move.l  SCdevtab(a2),a2 ; (A2) = device table
        cmp.w   (a2)+,d0        ; Is unit number valid?
        bgt.s   DcmRtrn         ; No, return (IOresult = 2)
        move.w  d0,d6           ; Compute offset into Unit Table
        mulu    #UTlen,d6       ; *
        adda.w  d6,a2           ; Get pointer to Unit Table entry
        move.l  #DcmPort,a1     ; Get slot 6 UART pointer
        move.b  UTslt(a2),d6    ; Get slot number (6-7)
        ext.w   d6              ; *
        subq.w  #6,d6           ; Compute UART pointer for slot
        lsl.w   #5,d6           ; *
        adda.w  d6,a1           ; *
        clr.w   d7              ; Clear IORESULT
        lea     DcmTABL,a2      ;
        lsl.w   #1,d4           ; D4 to word count
        move.w  0(a2,d4.w),d4   ; D4 = dist from DcmTABL
        jsr     0(a2,d4.w)      ; Go to appropriate driver
DCMRtrn movem.l (sp)+,d1-d6,a0-a6 ; Restore those registers!
        rts

DcmTABL data.w  DcmINST-DcmTABL ; Internal routine jump table
```

```
        data.w  DcmRD-DcmTABL
        data.w  DcmWR-DcmTABL
        data.w  DcmCLR-DcmTABL
        data.w  DcmBSY-DcmTABL
        data.w  DcmST-DcmTABL
        data.w  DcmUNMT-DcmTABL

;
; DcmINST
;
DcmINST rts                     ; Return
;
; DcmUNMT
;
DcmUNMT rts                     ; Return
;
; DcmST
;
DcmST   rts                     ; Return
;
; DcmRD
;
DcmRD
CrdLoop subq.w  #1,d2           ; More to read?
        bmi.s   CrdExit         ; No.
                                ;
CrdBusy btst    #RdBit,Ust(a1)  ; Is char in UART?
        boff.s  CrdBusy         ; No. Try again.
        move.b  Uda(a1),(a0)+   ; Yes. Fetch next character.
        bra.s   CrdLoop         ;
                                ;
CrdExit rts                     ; Return
        page
;
; DcmWR
;
DcmWR
CwrLoop subq.w  #1,d2           ; More to write?
        bmi.s   CwrExit         ; No.
                                ;
        btst    #RdBit,Ust(a1)  ; Is char in UART?
        boff.s  Cwr2            ; No, output next character
        move.b  Uda(a1),D0      ; Get character
        cmpi.b  #cs,D0          ; Stop output?
        bne.s   Cwr2            ; No, ignore character
                                ;
Cwr1    btst    #RdBit,Ust(a1)  ; Is char in UART?
        boff.s  Cwr1            ; No, wait some more
        move.b  Uda(a1),D0      ; Get character
        cmpi.b  #cq,D0          ; Start output?
        bne.s   Cwr1            ; No, ignore character
```

```
                                        ;
Cwr2    move.b  (A0)+,d0                ; Get next character.
        bsr.s   CrtOut1                 ; Output character
        cmpi.b  #cr,d0                  ; Was it a <CR>?
        bne.s   Cwr3                    ; No, go on
        btst    #LfBit,d5               ; Suppress line feed insertion?
        bne.s   Cwr3                    ; Yes, bypass <LF> insertion
        moveq   #lf,d0                  ; Add a <LF>
        bsr.s   CrtOut1                 ; Output character
Cwr3    bra.s   CwrLoop                 ; Check if finished with output
                                        ;
CwrExit rts                             ; Return
;
; CrtOut1
;
; This routine preserves all registers
;
CrtOut1
CwrBusy btst    #WrBit,Ust(a1)          ; Is UART output busy?
        boff.s  CwrBusy                 ; Yes. Try again.
        move.b  d0,Uda(a1)              ; Output the character
        rts                             ; Return
        page
;
; DcmCLR
;
DcmCLR  rts                             ; Return
;
; DcmBSY
;
; Returns: D0.B - Result
;
DcmBSY  moveq   #0,d0                   ; Assume FALSE (no character read
        btst    #RdBit,Ust(a1)          ; Character to read?
        boff.s  DcmBSYr                 ; No, return
        moveq   #1,d0                   ; Set TRUE
DcmBSYr rts                             ; Return

        page
        end                             ; DrvDtaCom
```

Standard Drivers | 5

This chapter deals with the operation of the standard CCOS drivers. There are two types of standard drivers: resident drivers, which are always present in the system, and loadable drivers, which are loaded after the boot process is omplete. Corvus-supplied loadable driver files have names that are prefixed with a "DRV." and reside on the CCSYS volume.

Most driver operations are implemented in the extensive functions in the CCLIB library (see "System Library Users Guide"). Before performing any exotic programming, it's a good idea to check and see if what you are trying to do hasn't already been done.

Direct communication with drivers is accomplished with the Pascal unit I/O mechanism. Specifically, UnitRead is used to retrieve data from the driver, UnitWrite is used to send data to the driver, UnitInstall initializes the driver, UnitClear flushes any buffers the driver may maintain, and UnitBusy is used to determine whether or not the driver has any data available. The remaining unit I/O procedure, UnitStatus, has special significance.

Special Functions

The UnitStatus call is used primarily for those functions which are not generic to all drivers. For example, setting the interleave and skew is a function that has meaning only in the context of a disk driver.

The special functions available to each driver will be described in the section dealing with that driver. The form of a UnitStatus call (from Pascal) is:

```
UnitStatus(UnitNumber,Buffer,FunctionCode);
```

where "UnitNumber" is the Pascal device number assigned to the device in question, "FunctionCode" is a driver dependent code for the action to be performed, and "Buffer" is a pointer to an arbitrary data structure which may be used either to pass information to the driver or receive information from it. UnitNumber and FunctionCode are always integers, while Buffer is always a VAR parameter (i.e. its address is passsed).

The data structure that Buffer points to will be described for each driver using Pascal syntax. Simple data structures will be represented as "word" or other base data type, while more complex structures will be shown as record declarations.

UnitStatus calls may also be made from a machine language program. See chapter 3, "Making System Calls", for details.

Driver Overview

Drivers are the software routines that control physical devices under CCOS. At boot time, CCOS loads its drivers from the boot volume and assigns them to the various physical devices on the system. This assignment process may also be performed manually with the ASSIGN utility.

A one-to-one correspondence between devices and drivers should not be assumed. A single driver routine may handle more than one physical device: for example, the DTACOM driver is responsible for both RS-232 ports. In addition, a single physical device may be addressed by more than one driver. For example, the second RS-232 port may be controlled by both the printer driver and DTACOM driver.

When a single driver controls more than one physical device, it is accessed as though there were a separate driver for each device. There is only one driver to control a local hard disk, although each volume on the disk will appear as a seperate device.

Direct communication with drivers is accomplished with the low-level unit I/O routines: UNITREAD, UNITWRITE, UNITBUSY, UNITSTATUS, and UNITCLEAR. Note that not all drivers support all unit routines! However, all drivers do support UNITMOUNT and UNITUNMOUNT, which are used to associate drivers with units.

To communicate with a driver you must know the unit number CCOS has assigned to it. The basic drivers the system has to work with are as follows:

Driver	What it does
DRV.ADISK	The (read only) Apple floppy disk driver.
DRV.DISPHZ and DRV.DISPVT	The display drivers, one for each screen oritentation.
DRV.DTACOM	The datacomm driver for the RS-232 ports.
DRV.EPRNT	The enhanced printer driver.
DRV.FDISK	The 8" floppy disk driver.
DRV.KYBD	The keyboard driver.

DRV.SYSTRM	Keyboard and display. Uses DRV.KYBD and DRV.DISPxx
DRV.TIMER	The timer driver.
DRV.DISPVT	The vertical display driver.
DRV.DISPHZ	The horizontal display driver.

In addition to these drivers, (which are referred to as loadable drivers) there are several resident drivers that are bound into the CCOS kernel. These are the SLOTIO, LOCAL (disk) and OMNINET drivers.

As mentioned above, drivers are assigned to unit numbers when CCOS is booted. The standard unit assignments are:

Unit		Driver
0		NULL device
1,2		DRV.SYSTRM
4		Boot volume
5		Default volume
6		DRV.PRNTR
9 - ?		DRV.ADISK, DRV.FDISK
? - (MaxDev-7)		LOCAL, OMNINET, or user
MaxDev - 6	(30)	SLOTIO
MaxDev - 5	(31)	DRV.DTACOM
MaxDev - 4	(32)	DRV.DTACOM
MaxDev - 3	(33)	OmniNet
MaxDev - 2	(34)	DRV.TIMER
MaxDev - 1	(35)	DRV.KYBD
MaxDev	(36)	DRV.DISPxx (may be DISPHZ or DISPVT)

MaxDev is the largest device number supported by the operating system, and is set to 36 in CCOS version 1.1a. The DTACOM, OMNINET, TIMER, KYB, and DISPLAY drivers are always loaded starting at MaxDev and working down, and this will remain constant in future releases of the system. Programs written to use these devices should use the standard library routines to determine the device number, as hardcoding the the device number may mean that the program will not work correctly on future operating system releases.

Some of the device assignments are not fixed. This is especially true in the case of disks (units 4,5, and 9 through MaxDev - 7), which may be local disks (Apple or 8 inch), local hard disks, or Omninet disks. Unit 4 is always the

volume you boot from, and unit 5 is generally the default volume. These units might be floppies, local disks, or Omninet, so the physical device assigned to these units will vary.

If the boot is performed from a hard disk, any floppy disks will be mounted starting on unit 9 and working up.

Units 3,7, and 8 are unassigned and may be used for user devices. Additionally, any of the disk units that are not in use may be assigned to a user device.

The Console/Systerm Driver

The console/systerm driver (DRV.SYSTRM) is responsible for the keyboard and display I/O of the Concept under CCOS. It addresses two separate drivers: DRV.KYBD and either the vertical display driver DRV.DISPVT or the horizontal display driver DRV.DISPHZ.

The console and systerm appear as units 1 (console) and 2 (systerm). Although DRV.SYSTRM is used for both devices, there are two important operational differences:

1. UNITREADs from /CONSOLE will be echoed to the screen, while UNITREADs from /SYSTERM will not;

2. UNITREADing an escape character (ASCII $1B) from /SYSTERM will return TWO escapes; UNITREADing an escape from /CONSOLE will not.

The KYBD Driver

Reading characters from the keyboard driver is normally done with the standard READ routine in whatever high level language is being used. UNITREADs may also be employed where a finer control is desired.

The KYBD driver recognizes three special UNITSTATUS calls:

```
Function code : 0                    Name: Send Raw
StatusRecord  : 1 word
```

This call sets or clears the SEND RAW flag in the driver. If the StatusRecord parameter is 0, then flag is cleared; if it is 1 then the flag is set.

If the SEND RAW flag is set, the keyboard driver will send all characters to the user, including control-S and control-Q.

```
Function code : 1                    Name: Break flag
StatusRecord  : 1 word
```

This call returns the value of the BREAK flag (in StatusRecord), then clears it. If the StatusRecord is 0 on return, the flag was clear. If it is 1 then the flag was set.

The BREAK flag represents the status of the Break key on the Concept keyboard. If it is set, then the Break key has been pressed since the last time this function was called. The Break flag is reset upon unit initialization of the keyboard driver or the execution of this call.

```
Function code : 2                    Name: Chars available
StatusRecord  : 1 word
```

This function returns the number of characters waiting in the keyboard input buffer.

EXAMPLE: The following Pascal fragment illustrates the use of function code 0 to set the SEND RAW flag:

```
CONST
  KEYB = 36;                         { KEYB driver unit }

VAR
  StatusCode : Integer;              { Occupies 1 word }
```

```
BEGIN
  StatusCode := 1;                    { Set "SEND RAW"  }
  UnitStatus(KEYB,StatusCode,1);      { Perform call    }
```

The Keyboard Translation Tables

The keyboard driver receives keyboard characters after they have been altered by the keyboard translation tables. These tables are used by the keyboard driver to generate the character sequences corresponding to the key pressed by the user. If a different set of key caps are used or a different set of character codes are desired then new Translation Tables must be built and loaded into the system.

The keyboard is connected to the computer by a transmission line. Through the line, the keyboard sends keycodes describing which key has been pressed or released. These keycodes, in conjunction with the Translation Tables, are used to generate the character sequences produced by the keyboard driver. Some keys, like the Shift key, affect which characters are generated when other keys are pressed. Some keys cause character sequences to be generated. What happens when a key is pressed or release is determined by the Translation Tables.

The Concept keyboard sends one byte of data, the keycode, when a key is pressed, and another byte when the key is released. The two bytes sent differe in the most significant bit: when the key is pressed, the MSB is set to 1; when it is released, the keycode's MSB is clear. The keycode generated is used as an index into the keyboard translation tables to generate the character(s) that the keyboard driver will return.

In order to build the Translation Tables a keycode map is needed. This map shows the keycode values for every key on the keyboard. Figure 1 is a keycode map for the current keyboard (Version 04, Selectric (R) style keyboard). Normally, the key caps show which character is generated for each keycode transmitted to the keyboard driver. Figure 2 is a key cap map for this same keyboard.

Version 04 keyboard key caps have either 1 or 2 symbols on them. A single symbol key cap specifies that the character is the same when it is either shifted or unshifted, except for the alphabet characters which get lower case if unshifted. Key caps with two symbols have the character for the lower symbol when unshifted and the character for the upper symbol when shifted.

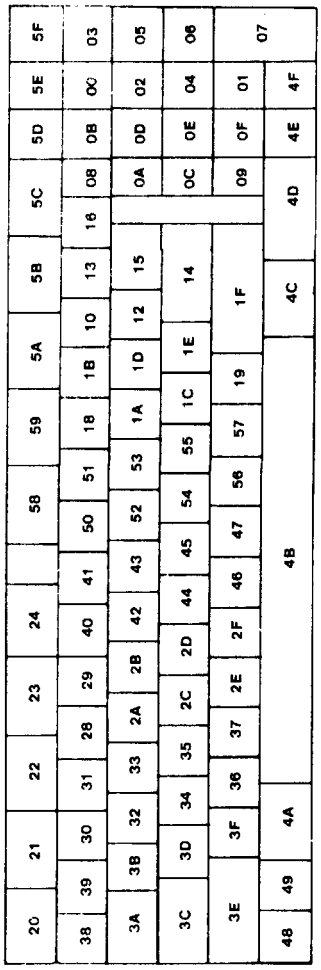

5F
03
5E
00
02
04
01
4F
5D
08
0D
0E
0F
4E
5C
08
0A
0C
09
4D
16
5B
13
15
14
1F
4C
10
12
5A
1B
1D
1E
19
59
18
1A
1C
57
58
51
53
55
56
50
52
54
47
24
41
43
45
46
40
42
44
2F
23
29
2B
2D
2E
22
28
2A
2C
37
31
33
35
36
21
30
32
34
3F
39
3B
3D
4A
20
38
3A
3C
3E
49
48
4B

Figure 1

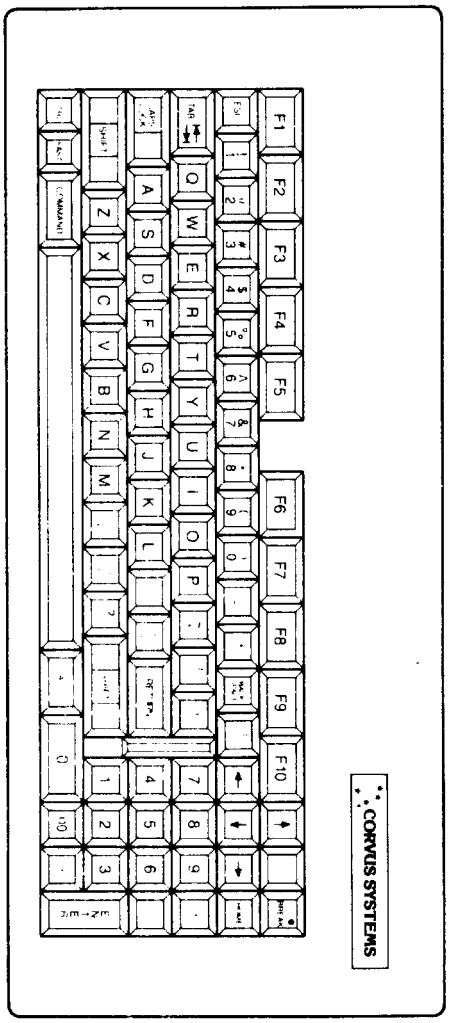


Figure 2

The translation tables must be defined in an assembly language program, such as the program CSK.REV4.TEXT listed in section 6.0. This program is actually a group of tables. The first table is TRANTBL which points to seven the translation tables.

The seven entries in this table point to the translation tables in the following order:

1) Shift Table (STABLE)
2) Regular table (RLTABLE)
3) Escape # sequence table (ETABLE)
4) Standard multiple character sequence table (SMTABLE)
5) Caps qualifier flag table (CQTABLE)
6) Release table (RLTABLE)
7) Break keycode table (BKEYCOD)

These entries must be in the above order.

The Shift Table

This table contains one byte for each keycode $00 - $5F. The byte is normally the character code for the specified keycode when the SHIFT key is depressed. Four special byte values are
used:

9E - use standard multiple character sequence table
9F - use caps qualifier flag table
9D - use escape # sequence table
00 - no character for this keycode.

The Regular Table

This table contains one byte for each keycode $00 - $5F. The byte is normally the charater code for the specified keycode when the SHIFT key is not depressed. Four special byte values are used:

9E - use standard multiple character sequence table
9F - use caps qualifier flag table
9D - use escape # sequence table
00 - no character for this keycode.

The Escape Sequence Table

This table is used when a table code of $9D is found in key closure or a table code of $9D is found in key shift table

(STABLE) or the regular table (RLTABLE). It specifies a key which has an ESC # character sequence. Each keycode may have a different character based on the state of the two qualifier keys (SHIFT and COMMAND).

Each table entry has the form (entry length = 10 bytes) :

1) Keycode (1 byte).
2) filler byte : its value is 0 (1 byte).
3) UnSHIFTed & UnCOMMANDed (2 bytes).
4) SHIFT only (2 bytes).
5) COMMAND only (2 bytes).
6) COMMAND & SHIFT together (2 bytes).

Values for the version 04 keyboard:

KEYCODE	FILL	US/UC	S only	C only	C/S	KEY NAME
$20	00	00	0A	14	1E	Function key 1
$21	00	01	0B	15	1F	Function key 2
$22	00	02	0C	16	20	Function key 3
$23	00	03	0D	17	21	Function key 4
$24	00	04	0E	18	22	Function key 5
$4A	00	FF	FF	FF	FF	COMMAND (closure)
$58	00	05	0F	19	23	Function key 6
$59	00	06	10	1A	24	Function key 7
$5A	00	07	11	1B	25	Function key 8
$5B	00	08	12	1C	26	Function key 9
$5C	00	09	13	1D	27	Function key 10
$CA	00	FE	FE	FE	FE	COMMMAND (release)

The Standard Multiple Character Sequence Table

This table is used on key closure when a $9E table code is in the shift table (STABLE) or regular table (RLTABLE). Every entry with a $9E table code in the STABLE or RLTABLE must be in this table.

Each entry is composed of 3 fields. 1) the keycode, 2) the string length, and 3) the actual string. The string is the sequence of character codes placed in the buffer for this key. The Table does not have to be in keycode order. The table ends with a special keycode of $FF and length of 0.

Values for the version 04 keyboard:

KEYCODES	STRING LENGTH	STRING
$00 (cursor right)	2	$1B $43 (esc C)
$03 (HOME up)	2	$1B $48 (esc H)
$07 (enter)	2	$1B $64 (esc d)
$08 (cursor left)	2	$1B $44 (esc D)
$0B (cursor down)	2	$1B $42 (esc B)
$3A (back tab)	2	$1B $69 (esc i)
$5D (cursor up)	2	$1B $41 (esc A)
$4E (double zero)	2	$30 $30 (00)
$FF	0	END OF THE TABLE

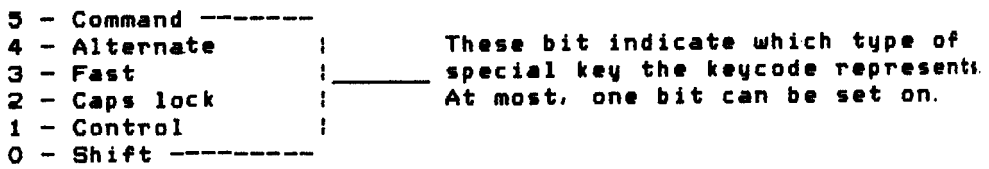
The Caps Lock and Qualifier Flag Table

This table contains one byte for each keycode $00 - $5F. The keycode is a direct index into the table. Each byte is a set of flags. All unused bits must be cleared (value = 0). The high order bit is the caps lock flag for the corresponding Keycode. If the bit is set, this keycode generates a shifted character if the CAPS LOCK key is locked. Bit 6 is a special COMMAND key flag. The remaining bits are special key qualifier flags.

The bits currently defined are :

7 - Caps lock flag : when set means this keycode generates a shifted character when Caps lock is locked.

6 - Special COMMAND key flag:
0 uses ETABLE for closure - keycode high order bit closure.
0 uses ETABLE for release - keycode has high order bit set.
0 special non-repeating key.



The values for the version 04 keyboard are listed in the attached program CSK.REV4.TEXT.

The Release Table

This table specifies which keycodes have an action on key release. Each table has 2 fields. 1) the keycode, and 2) the action code.

The action code has 3 possible value types. If the action code is $9D it specifies a key with a escape # sequence table (ETABLE) entry. If the action code is $9E it specifies a qualifier keycode. Any other action code is a character code to be placed into the buffer. The end of the table is specified by a special keycode of $FF and an action code of $00.

Values for the version 04 keyboard:

KEYCODE	ACTION CODE	KEY NAME
$1F	$9E	Right SHIFT
$3C	$9E	CAPS LOCK
$3E	$9E	Left SHIFT
$48	$9E	Control (CTRL)
$49	$9E	FAST
$4A	$9E	COMMAND
$4C	$9E	Alternate (ALT)
$FF	$00	NULL keycode - END OF TABLE

The Break Key Code Table

This table consists of one byte. It is the Keycode for the key which performs the start/stop toggle. The value for the version 04 keyboard is : $DF. This is the keycode for BREAK closure.

Translation Table Examples

Single characters

This section gives the user several examples of how to change the keyboard tranlation tables. The examples deal with the unmarked key on the top row of keys (keycode $5E).

The first example is to use the unmarked key (keycode $5E) as a standard alphabetic character key. This involves setting a value in the translation tables for the unshifted, shifted, and qualifier cases of the key.

A. These tables use the keycode value as an offset into the tables. Locate the unmarked key on the keyboard and note the position. Locate the same key in the keycode chart and note the keycode for closure (5E).

B. For this example let us assume the desired output of the Translations Tables is to be the alphabetic character 't' for unSHIFTed, 'T' for SHIFTed, and 'T' for CAPS LOCK.

C. Create a file with the same tables as the program CSK.REV4.TEXT.

D. Locate the position 5E in the SHIFT Table. Note that the current entry is 9F hex, which indicates the key is a qualifier. In this example the SHIFT Table entry will be changed to a 'T' or 54 hex. Edit the STABLE at postion 5E hex to contain the value 54 hex.

```
The shift table is indexed by keycode. Each byte represents
the character code for the corresponding keycode.

The character symbol is above each character code
   SMC = special value for Standard Multiple Character Sequence   ($9E)
   GUL = special value for Qualifier                              ($9F)
   EST = special value for Escape Sharp Character Sequence        ($9D)
   ... = No key for this keycode
```

STABLE

```
         SMC  3   9   SMC  6   ,   -   cr  SMC  1   7   SMC  4   8   5   2   ;MSB
DATA. B  $9E, $33, $39, $9E, $36, $2C, $2D, $0D, $9E, $31, $37, $9E, $34, $38, $35, $32 ; $00
          +   ...  (   del  cr  )   !   ...  )   ?   P   _   :   ~   "   GUL
DATA. B  $2B, $00, $7B, $7F, $0D, $7D, $7C, $00, $29, $3F, $50, $5F, $3A, $7E, $22, $9F ; $10
         EST EST EST EST EST ... ... ... $   %   R   T   F   G   V   B
DATA. B  $9D, $9D, $9D, $9D, $9D, $00, $00, $00, $24, $25, $52, $54, $46, $47, $56, $42 ; $20
          @   #   W   E   S   D   X   C   esc  !  SMC  Q  GUL  A  GUL  Z
DATA. B  $40, $23, $57, $45, $53, $44, $58, $43, $1B, $21, $9E, $51, $9F, $41, $9F, $5A ; $30
          ^   &   Y   U   H   J   N   M  GUL GUL GUL sp  GUL  O  SMC
DATA. B  $5E, $26, $59, $55, $48, $4A, $4E, $4D, $9F, $9F, $9F, $20, $9F, $30, $9E, $2E ; $40
          *   (   I   O   K   L   <   >  EST EST EST EST EST SMC GUL GUL
DATA. B  $2A, $28, $49, $4F, $4B, $4C, $3C, $3E, $9D, $9D, $9D, $9D, $9D, $9E, $9F, $9F ; $50
LSB       0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

```
            Change the last line to the following ----------------+
                                                                  |
                                                                  V
         *   (   I   O   K   L   <   >  EST EST EST EST EST SMC  T  GUL
DATA. B  $2A, $28, $49, $4F, $4B, $4C, $3C, $3E, $9D, $9D, $9D, $9D, $9D, $9E, $54, $9F ; $50
LSB       0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

E. Locate the position 5E in the REGULAR Table. Note that the current entry is 9F hex which indicates the key is a qualifier. In this example the REGULAR table entry will be changed to a 't' or 74 hex. Edit the RTABLE at position 5E hex to contain the value 74 hex.

RTABLE (The Regular Table)

```
         SMC  3   9   SMC  6   ,   -   cr  SMC  1   7   SMC  4   8   5   2   ;MSB
DATA. B  $9E, $33, $39, $9E, $36, $2C, $2D, $0D, $9E, $31, $37, $9E, $34, $38, $35, $32 ; $00
          =   ...  [   bs   cr  ]   \   ...  0   /   p   -   ;   `   '   GUL
DATA. B  $3D, $00, $5B, $08, $0D, $5D, $5C, $00, $30, $2F, $70, $2D, $3B, $60, $27, $9F ; $10
         EST EST EST EST EST ... ... ... 4   5   r   t   f   g   v   b
DATA. B  $9D, $9D, $9D, $9D, $9D, $00, $00, $00, $34, $35, $72, $74, $66, $67, $76, $62 ; $20
          2   3   w   e   s   d   x   c   esc  1  SMC  q  GUL  a  GUL  z
DATA. B  $32, $33, $77, $65, $73, $64, $78, $63, $1B, $31, $09, $71, $9F, $61, $9F, $7A ; $30
```

```
          6   7   y   u   h   j   n   m  GUL GUL GUL sp  GUL  O  SMC  .
DATA. B $36,$37,$79,$75,$68,$6A,$6E,$6D,$9F,$9F,$9F,$20,$9F,$30,$9E,$2E ;$40
          8   9   i   o   k   l   ,   .  EST EST EST EST EST SMC GUL GUL
DATA. B $38,$39,$69,$6F,$6B,$6C,$2C,$2E,$9D,$9D,$9D,$9D,$9D,$9E,$9F,$9F ;$50
LSB       0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

Change the last line to the following:

```
          8   9   i   o   k   l   ,   .  EST EST EST EST EST SMC  t  GUL
DATA. B $38,$39,$69,$6F,$6B,$6C,$2C,$2E,$9D,$9D,$9D,$9D,$9D,$9E,$74,$9F ;$50
LSB       0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

F. Locate the position 5E in the CAPS/QUALIFIER Table. Note that the current entry is 00 hex which indicates the key does not have any flags set in the CAPS/QUALIFIER Table. In this example the CAPS/QUALIFIER Table entry will be changed to a 80 hex, to set the Caps lock flag in the table. Edit the CQTABLE at position 5E hex to contain the value 80 hex.

Each byte has 8 flags :

D7 = Caps lock flag : when set means this keycode generates a shifted character when the Caps lock qualifier flag is set.

D6 = Qualifier has an ESC # sequence flag. When set then must process the keycode as a non-repeating ESC # sequence. Also has a Release sequence.

```
D5 = Command --------
D4 = Alternate       |
D3 = Fast            |      This bit says which type of Qualifier
D2 = Caps lock       |_____ key the Keycode represents.
D1 = Control         |
D0 = Shift  _________|
```

```
CQTABLE (The caps/qualifier table)
                                                                          ;MSB
DATA. B $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;$00
DATA. B $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$80,$00,$00,$00,$00,$01 ;$10
DATA. B $00,$00,$00,$00,$00,$00,$00,$00,$00,$00,$80,$80,$80,$80,$80,$80 ;$20
DATA. B $00,$00,$80,$80,$80,$80,$80,$80,$00,$00,$00,$80,$04,$80,$01,$80 ;$30
DATA. B $00,$00,$80,$80,$80,$80,$80,$80,$02,$08,$60,$00,$10,$00,$00,$00 ;$40
DATA. B $00,$00,$80,$80,$80,$80,$00,$00,$00,$00,$00,$00,$00,$00,$00,$00 ;$50
 LSB       0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

Change the last line to the following:

```
DATA. B $00,$00,$80,$80,$80,$80,$00,$00,$00,$00,$00,$00,$00,$00,$80,$00 ;$50
```

LSB	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Q. Save the edited version of the Keyboard Translation Tables to a test file. Assemble the file as follows:

ASM68K filename [RETURN]

Upon completion of the assembly, link the file for quick load as follows:

LINKER filename [RETURN]

The last step is to load the new Keyboard Translation Table.

Press [WndowMgr].

Press [LdKybdCh].

Enter the filename, [RETURN].

A successful load of the tables will be noted in the Command Line. Begin testing the results of the new tables by pressing the unmarked key. Use the SHIFT key and the CAPS LOCK key and note the results.

Translation Table Examples
Multiple character sequences

This example deals with the modification of the standard multiple character sequence table. In this example, the translation tables will be modified so that the unmarked key (keycode $5E) will be treated as cursor right.

A. Create a file with the same entries as the CSK.REV4.TEXT file.

B. Locate the standard multiple character sequence table within the file. It should be as follows:

Standard multiple character sequence table

FORMAT : (KEYCODE, LENGTH, CHARACTER_SEQUENCE)

The LENGTH field is the number of characters in the CHARACTER SEQUENCE field. The CHARACTER SEQUENCE is the characters to return for the Keycode.

```
SMTABLE
        KEYCODE LENGTH CHARACTER SEQUENCE
DATA.B    $00,    2,     $1B,$43                ;CURSOR RIGHT
DATA.B    $03,    2,     $1B,$48                ;HOME UP
DATA.B    $07,    2,     $1B,$64                ;ENTER
DATA.B    $08,    2,     $1B,$44                ;CURSOR LEFT
DATA.B    $0B,    2,     $1B,$42                ;CURSOR DOWN
DATA.B    $3A,    2,     $1B,$69                ;BACK TAB
DATA.B    $5D,    2,     $1B,$41                ;CURSOR UP
DATA.B    $4E,    2,     $30,$30                ;DOUBLE ZERO-( 00 KEY )
DATA.B    $FF,    0                             ;NULL KEYCODE - END OF TABL
```

C. Enter a duplication of the first entry in the table as the last entry in the table Change the KEYCODE from $00 to $5E. The unmarked key is now defined as CURSOR RIGHT.

```
SMTABLE
        KEYCODE LENGTH CHARACTER SEQUENCE
DATA.B    $00,    2,     $1B,$43                ;CURSOR RIGHT
DATA.B    $03,    2,     $1B,$48                ;HOME UP
DATA.B    $07,    2,     $1B,$64                ;ENTER
DATA.B    $08,    2,     $1B,$44                ;CURSOR LEFT
DATA.B    $0B,    2,     $1B,$42                ;CURSOR DOWN
DATA.B    $3A,    2,     $1B,$69                ;BACK TAB
DATA.B    $5D,    2,     $1B,$41                ;CURSOR UP
DATA.B    $4E,    2,     $30,$30                ;DOUBLE ZERO-( 00 KEY )
DATA.B    $5E,    2,     $1B,$43                ;CURSOR RIGHT
DATA.B    $FF,    0                             ;NULL KEYCODE - END OF TABL
```

D. Locate the position 5E in the shift table. Note that the current entry is 9F hex which indicates the key is a qualifier. In this example the shift table entry will be changed to a $9E hex. Edit the STABLE at postion 5E hex to contain the value 9E hex.

```
The character symbol is above each character code
   SMC = special value for Standard Multiple Character Sequence   ($9E)
   QUL = special value for Qualifier                              ($9F)
   EST = special value for Escape Sharp Character Sequence        ($9D)
   ... = No key for this keycode

STABLE (The shift table)

          SMC  3   9   SMC  6   ,   -   cr  SMC  1   7   SMC  4   8   5   2   ;MSB
DATA.B    $9E,$33,$39,$9E,$36,$2C,$2D,$0D,$9E,$31,$37,$9E,$34,$38,$35,$32 ;$00
           +   ...  {   del  cr  }   :   ...  )   ?   P   _   :   ~   "   QUL
DATA.B    $2B,$00,$7B,$7F,$0D,$7D,$7C,$00,$29,$3F,$50,$5F,$3A,$7E,$22,$9F ;$10
          EST EST EST EST EST ... ... ... $   %   R   T   F   G   V   B
DATA.B    $9D,$9D,$9D,$9D,$9D,$00,$00,$00,$24,$25,$52,$54,$46,$47,$56,$42 ;$20
           @   #   W   E   S   D   X   C  esc  !  SMC  Q  QUL  A  QUL  Z
DATA.B    $40,$23,$57,$45,$53,$44,$58,$43,$1B,$21,$9E,$51,$9F,$41,$9F,$5A ;$30
           ^   &   Y   U   H   J   N   M  QUL QUL QUL sp  QUL  O  SMC  .
DATA.B    $5E,$26,$59,$55,$48,$4A,$4E,$4D,$9F,$9F,$9F,$20,$9F,$30,$9E,$2E ;$40
           *   (   I   O   K   L   <   >  EST EST EST EST EST SMC QUL QUL
DATA.B    $2A,$28,$49,$4F,$4B,$4C,$3C,$3E,$9D,$9D,$9D,$9D,$9D,$9E,$9F,$9F ;$50
LSB        0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

Change the last line to the following:

```
           *   (   I   O   K   L   <   >  EST EST EST EST EST SMC SMC QUL
DATA.B    $2A,$28,$49,$4F,$4B,$4C,$3C,$3E,$9D,$9D,$9D,$9D,$9D,$9E,$9E,$9F ;$50
LSB        0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

E. Locate the position 5E in the regular table. Note that the current entry is 9F hex which indicates the key is a qualifier. In this example the regular table entry will be changed to a 9E hex. Edit the RTABLE at position 5E hex to contain the value 9E hex.

```
The character symbol is above each character code
   SMC = special value for Standard Multiple Character Sequence   ($9E)
   QUL = special value for Qualifier                              ($9F)
   EST = special value for Escape Sharp Character Sequence        ($9D)
```

```
    ... = No key for this keycode

RTABLE (The regular table)

          SMC  3   9   SMC  6   ,   -   cr  SMC  1   7   SMC  4   8   5   2   ;MSB
DATA.B    $9E, $33, $39, $9E, $36, $2C, $2D, $0D, $9E, $31, $37, $9E, $34, $38, $35, $32 ,$00
           =   ...  [   bs   cr   ]   \   ...  0   /   p   -   ;   `   '   QUL
DATA.B    $3D, $00, $5B, $08, $0D, $5D, $5C, $00, $30, $2F, $70, $2D, $3B, $60, $27, $9F ;$10
          EST EST EST EST EST ... ... ... 4   5   r   t   f   g   v   b
DATA.B    $9D, $9D, $9D, $9D, $9D, $00, $00, $00, $34, $35, $72, $74, $66, $67, $76, $62 ;$20
           2   3   w   e   s   d   x   c  esc  1  SMC  q  QUL  a  QUL  z
DATA.B    $32, $33, $77, $65, $73, $64, $78, $63, $1B, $31, $09, $71, $9F, $61, $9F, $7A ;$30
           6   7   y   u   h   j   n   m  QUL QUL QUL sp  QUL  0  SMC  .
DATA.B    $36, $37, $79, $75, $68, $6A, $6E, $6D, $9F, $9F, $9F, $20, $9F, $30, $9E, $2E ;$40
           8   9   i   o   k   l   ,   .  EST EST EST EST EST SMC QUL QUL
DATA.B    $38, $39, $69, $6F, $6B, $6C, $2C, $2E, $9D, $9D, $9D, $9D, $9D, $9E, $9F, $9F ,$50
LSB        0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F

          Change the last line to the following:

           8   9   i   o   k   l   ,   .  EST EST EST EST EST SMC SMC QUL
DATA.B    $38, $39, $69, $6F, $6B, $6C, $2C, $2E, $9D, $9D, $9D, $9D, $9D, $9E, $9E, $9F ;$50
LSB        0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F

F.  Save, Assemble, Link, and Load as in the previous example.
```

Translation Table Examples
Escape sharp sequences

This example deals with the modification of the escape sharp sequence table. The translation tables will now be modified to use the unmarked key as the function key 1.

A. Create a file with the same entries as the CSK.REV4.TEXT file.

B. Locate the escape sharp sequence table within the file. It should be as follows:

```
 ESCAPE SHARP(#) SEQUENCE TABLE
        FORMAT :   (KEYCODE,FILLER,US/UC,SHIFT,COMMAND,C/S)
```

The fill field is added to keep each record on an even byte boundary The other fields contain the character sequence to follow the ESCAPE # characters:

```
   US/UC    = when the Shift and Command key are released
   SHIFT    = when only the Shift key is still being pressed
   COMMAND  = when only the Command key is still being pressed
   C/S      = when the Shift and Command keys are still
              being pressed
```

```
ETABLE  (The escape sharp sequence table)
          KEYCODE FILL    US/UC   SHIFT   COMMAND    C/S
  DATA.B  $20,    0,      '00',   '0A',   '14',      '1E' ;FUNCTION KEY 1
  DATA.B  $21,    0,      '01',   '0B',   '15',      '1F' ;FUNCTION KEY 2
  DATA.B  $22,    0,      '02',   '0C',   '16',      '20' ;FUNCTION KEY 3
  DATA.B  $23,    0,      '03',   '0D',   '17',      '21' ;FUNCTION KEY 4
  DATA.B  $24,    0,      '04',   '0E',   '18',      '22' ;FUNCTION KEY 5
  DATA.B  $4A,    0,      'FF',   'FF',   'FF',      'FF' ;LEFT COMMAND (CLOSURE)
  DATA.B  $58,    0,      '05',   '0F',   '19',      '23' ;FUNCTION KEY 6
  DATA.B  $59,    0,      '06',   '10',   '1A',      '24' ;FUNCTION KEY 7
  DATA.B  $5A,    0,      '07',   '11',   '1B',      '25' ;FUNCTION KEY 8
  DATA.B  $5B,    0,      '08',   '12',   '1C',      '26' ;FUNCTION KEY 9
  DATA.B  $5C,    0,      '09',   '13',   '1D',      '27' ;FUNCTION KEY 10
  DATA.B  $CA,    0,      'FE',   'FE',   'FE',      'FE' ;LEFT COMMAND (RELEASE)
```

C. Enter a duplication of the first entry in the table as the last entry in the table. Change the keycode from $20 to $5E. The unmarked key is now defined as function key 1.

```
ETABLE
          KEYCODE FILL    US/UC   SHIFT   COMMAND    C/S
  DATA.B  $20,    0,      '00',   '0A',   '14',      '1E' ;FUNCTION KEY 1
  DATA.B  $21,    0,      '01',   '0B',   '15',      '1F' ;FUNCTION KEY 2
  DATA.B  $22,    0,      '02',   '0C',   '16',      '20' ;FUNCTION KEY 3
```

```
DATA.B  $23,    0,    '03',  '0D',  '17',    '21' ;FUNCTION KEY 4
DATA.B  $24,    0,    '04',  '0E',  '18',    '22' ;FUNCTION KEY 5
DATA.B  $4A,    0,    'FF',  'FF',  'FF',    'FF' ;LEFT COMMAND (CLOSURE)
DATA.B  $58,    0,    '05',  '0F',  '19',    '23' ;FUNCTION KEY 6
DATA.B  $59,    0,    '06',  '10',  '1A',    '24' ;FUNCTION KEY 7
DATA.B  $5A,    0,    '07',  '11',  '1B',    '25' ;FUNCTION KEY 8
DATA.B  $5B,    0,    '08',  '12',  '1C',    '26' ;FUNCTION KEY 9
DATA.B  $5C,    0,    '09',  '13',  '1D',    '27' ;FUNCTION KEY 10
DATA.B  $5E,    0,    '00',  '0A',  '14',    '1E' ;FUNCTION KEY 1
DATA.B  $CA,    0,    'FE',  'FE',  'FE',    'FE' ;LEFT COMMAND (RELEASE)
```

D. Locate the position 5E in the shift table. Note that the current entry is 9F hex which indicates the key is a qualifier. In this example the shift table entry will be changed to a $9D hex. Edit the STABLE at postion 5E hex to contain the value 9D.

STABLE

```
        SMC  3    9    SMC  6    ,    -    cr   SMC  1    7    SMC  4    8    5    2    ;MSB
DATA.B  $9E, $33, $39, $9E, $36, $2C, $2D, $0D, $9E, $31, $37, $9E, $34, $38, $35, $32 ; $00
         +   ...  {   del  cr   }    !   ...   )    ?    P    _    :    ~    "   QUL
DATA.B  $2B, $00, $7B, $7F, $0D, $7D, $7C, $00, $29, $3F, $50, $5F, $3A, $7E, $22, $9F ; $10
        EST EST  EST  EST  EST  ...  ...  ...   $    %    R    T    F    G    V    B
DATA.B  $9D, $9D, $9D, $9D, $9D, $00, $00, $00, $24, $25, $52, $54, $46, $47, $56, $42 ; $20
         @    #    W    E    S    D    X    C   esc   !   SMC  G   QUL   A   QUL   Z
DATA.B  $40, $23, $57, $45, $53, $44, $58, $43, $1B, $21, $9E, $51, $9F, $41, $9F, $5A ; $30
         ^    &    Y    U    H    J    N    M   QUL  QUL  QUL  sp   QUL   O   SMC   .
DATA.B  $5E, $26, $59, $55, $48, $4A, $4E, $4D, $9F, $9F, $9F, $20, $9F, $30, $9E, $2E ; $40
         *    (    I    O    K    L    <    >   EST  EST  EST  EST  EST  SMC  QUL  QUL
DATA.B  $2A, $28, $49, $4F, $4B, $4C, $3C, $3E, $9D, $9D, $9D, $9D, $9D, $9E, $9F, $9F ; $50
LSB      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
```

Change the last line to the following:

```
         *    (    I    O    K    L    <    >   EST  EST  EST  EST  EST  SMC  EST  QUL
DATA.B  $2A, $28, $49, $4F, $4B, $4C, $3C, $3E, $9D, $9D, $9D, $9D, $9D, $9E, $9D, $9F    ; $
LSB      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
```

E. Locate the position 5E in the regular table. Note that the current entry is 9F hex which indicates the key is a qualifier. In this example the regular table entry will be changed to a $9D hex. Edit the RTABLE at position 5E hex to contain the value 9D hex.

RTABLE (The regular table)

```
          SMC  3   9   SMC  6   ,   -   cr  SMC  1   7   SMC  4   8   5   2   ;MSB
DATA.B    $9E,$33,$39,$9E,$36,$2C,$2D,$0D,$9E,$31,$37,$9E,$34,$38,$35,$32 ;$00
           =   ...  [   bs  cr  ]   \   ...  0   /   p   -   ;   `   '  QUL
DATA.B    $3D,$00,$5B,$08,$0D,$5D,$5C,$00,$30,$2F,$70,$2D,$3B,$60,$27,$9F ;$10
          EST EST EST EST EST ... ... ...  4   5   r   t   f   g   v   b
DATA.B    $9D,$9D,$9D,$9D,$9D,$00,$00,$00,$34,$35,$72,$74,$66,$67,$76,$62 ;$20
           2   3   w   e   s   d   x   c  esc  1  SMC  q  QUL  a  QUL  z
DATA.B    $32,$33,$77,$65,$73,$64,$78,$63,$1B,$31,$09,$71,$9F,$61,$9F,$7A ;$30
           6   7   y   u   h   j   n   m  QUL QUL QUL sp  QUL  0  SMC  .
DATA.B    $36,$37,$79,$75,$68,$6A,$6E,$6D,$9F,$9F,$9F,$20,$9F,$30,$9E,$2E ;$40
           8   9   i   o   k   l   ,   .  EST EST EST EST EST SMC QUL QUL
DATA.B    $38,$39,$69,$6F,$68,$6C,$2C,$2E,$9D,$9D,$9D,$9D,$9D,$9E,$9F,$9F ;$50
LSB        0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

Change the last line to the following:

```
          8   9   i   o   k   l   ,   .  EST EST EST EST EST SMC EST QUL
DATA.B    $38,$39,$69,$6F,$68,$6C,$2C,$2E,$9D,$9D,$9D,$9D,$9D,$9E,$9D,$9F ;$50
LSB        0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

F Save, Assemble, Link, and Load as in the previous example.

Making a default keyboard table

To make a keyboard translation table the system default table, merely transfer it to the system volume with the name CSK.DEFAULT.

The Display Driver

The display driver is normally controlled with the intrinsic output capabilities of the high level languages in use (i.e. WRITE and WRITELN in the case of Pascal). However, the UNITWRITE and UNITSTATUS functions can also be used.

Windows

The display driver's output is always to the current window. CCOS supports up to 10 windows on the screen at any one time. Normally, three windows are in use: the large system window, the two-line command window immediately below it, and the function key window containing the function key labels.

Windows can be created with the window manager or through a user program. Each window has a variety of attributes that are described in its window record. The SYSCOM field Current Window Record Pointer always points to the address of the record of the window currently in use.

The format of a window record is as follows:

```
     WndRcd    =  record
{length offset}
{    4      0 }   charpt:  pCharSet; {character set record pointer}
{    4      4 }   homept:  pBytes;   {home (upper left) pointer}
{    4      8 }   curadr:  pBytes;   {current location pointer}
{    2     12 }   homeof:  integer;  {bit offset of home location}
{    2     14 }   basex:   integer;  {home x value, rel to root window}
{    2     16 }   basey:   integer;  {home y value, rel to root window}
{    2     18 }   lngthx:  integer;  {maximum x value, bits rel to window}
{    2     20 }   lngthy:  integer;  {maximum y value, bits rel to wimdow}
{    2     22 }   cursx:   integer;  {current x value, bits rel to window}
{    2     24 }   cursy:   integer;  {current y value, bits rel to window}
{    2     26 }   bitofs:  integer;  {bit offset of current address}
{    2     28 }   grorgx:  integer;  {graphics - origin x, bits rel to home}
{    2     30 }   grorgy:  integer;  {graphics - origin y, bits rel to home}
{    1     32 }   attr1:   byte;     {inverse, underscore, insert}
{    1     33 }   attr2:   byte;     {v/h, graphics/char, cursor on/off,
                                      cursor inv/underline}
{    1     34 }   state:   byte;     {used for decoding escape sequences}
{    1     35 }   rcdlen:  byte;     {window description record length}
{    1     36 }   attr3:   byte;     {enhanced character set attributes}
{    1     37 }   fill1:   byte;     {currently unused}
{    1     38 }   fill2:   byte;     {currently unused}
{    1     39 }   fill3:   byte;     {currently unused}
{    4     40 }   fill4:   longint;  {currently unused}
{    4     44 }   wwsptr:  pBytes;   {window working storage pointer}
```

```
{ total    48 }   end;
```

Character sets

Each display window has an associated character set record that describes the size of the font in use, as well as the dot patterns of the individual characters. The format of the character set record is:

```
CharSet = Record
{ Length  Offset}
{   4       0   } TblLoc: PBytes;  {Char set data pointer }
{   2       4   } Lpch  : Integer; {Scan lines per char.  }
{   2       6   } Bpch  : Integer; {Bits per character    }
{   2       8   } FrstCh: Integer; {ASCII of 1st char.    }
{   2      10   } LastCh: Integer; {ASCII of last char.   }
{   4      12   } Mask  : LongInt; {Mask used in cell pos.}
{   1      16   } Attr1 : Byte    ; {Attributes...not used }
{   1      17   } Fill1 : Byte    ; {Not used              }
{ total    18   }
```

The Lpch and Bpch fields describe the height and width of the character cell in pixels. If the screen is vertical, Lpch describes the width of the character, and Bpch describes the height. If the screen is horizontal, the fields' meanings are reversed.

A character set file consists of four integers giving the cell width, cell height, 1st character ASCII code, and last character ASCII code, followed by an array[first..last] of CharData. Each element of CharData is an array[1..height] of Byte if the character width is less than or equal to 8, or array[1..height] of Integer if the character width is more than 8. Scanline data is left-justified within the data type.

Controlling the display

UNITWRITE can be used to send bytes to the display. However, the display driver uses many control and escape codes to activate its various features. A list of these codes follows:

Command Sequence	Hex Codes	Description
Ctl-G	07	bell
Ctl-H	08	cursor left (backspace)
Ctl-I	09	tab (8 spaces)
Ctl-J	0A	cursor down (linefeed)
Ctl-K	0B	cursor up
Ctl-L	0C	cursor right
Ctl-M	0D	carriage return
ESC = col row	1B,3D,col,row	gotoxy
ESC A	1B,41	cursor up
ESC B	1B,42	cursor down (linefeed)
ESC C	1B,43	cursor right
ESC D	1B,44	cursor left (backspace)
ESC E	1B,45	insert line
ESC G 0	1B,47,30	set video to normal
ESC G 4	1B,47,34	set video to inverse
ESC G 8	1B,47,38	set video to underline
ESC G <	1B,47,3C	set video to inverse+underline
ESC H	1B,48	cursor home
ESC J	1B,49	clear window, home cursor
ESC K	1B,4B	clear to end of line
ESC O c1 c2	1B,4F,c1,c2	overstrike c1, c2
ESC Q	1B,51	insert character
ESC R	1B,52	delete line
ESC W	1B,57	delete character
ESC Y	1B,59	clear to end of window
ESC Z	1B,60,N	invert N characters
ESC a	1B,61	page mode on
ESC b	1B,62	turn off cursor
ESC c	1B,63	turn on cursor
ESC d	1B,64	enter key: carriage return
ESC e	1B,65	character enhancements
ESC f	1B,66	*fill block
ESC g	1B,67	graphics mode
ESC i	1B,69	back tab (8 spaces)
ESC l	1B,6C	*draw line
ESC n	1B,6D	*copy block
ESC n	1B,6E	turn off scrolling
ESC o	1B,6F	*set graphics origin
ESC p	1B,70	*plot point
ESC q	1B,71	insert mode
ESC r	1B,72	insert mode off
ESC s	1B,73	turn on scrolling
ESC t	1B,74	text mode
ESC u	1B,75	underscore cursor
ESC v	1B,76	inverse cursor
ESC w	1B,77	wrap at end & beginning of line

ESC x	1B,78	no wrap at end & beg. of line
ESC y	1B,79	page mode off
ESC z	1B,7A	invert screen

* Graphics functions:

Function	Parameters	Byte count
Set origin	x,y,qualifier	7 (11221)
Plot point	x,y,mode	7 (11221)
Draw line	x1,y1,x2,y2,mode	11 (1122221)
Fill block	x,y,height,width,density	11 (1122221)
Copy block	x1,y1,height,width,x2,y2	14 (11222222)
WriteBytes	(see UnitStatus)	
ReadBytes	(see UnitStatus)	

mode: <0 invert, =0 clear, >0 set

density: 1 dense, 2 less dense, etc.

qualifier: 1 rel to graphics origin
2 abs graphics origin
3 rel to cursor position
4 abs text origin

These driver capabilities can easily be used directly from a program. For example, a Pascal program could invert the current window (the ESC-z sequence) with:

```
WRITE(Chr(27),'z');
```

Note: Control of most of the display driver's special capabilities is provided in the CCcrtIO unit in CCLIB.

The display driver also supports various character enhancements: overstriking, underline, double underline, superscripting, subscripting, strike out, and bold face.

The overstrike enhancement is specified as follows:

ESC O <char1> <char2>

<Char1> and <char2> are OR'ed together at the cursor position. If a character is not in the character set, a blank is substituted. Only two characters may be overstruck at one cursor position.

The other character enhancements are specified:

ESC e <byte>

<Byte> is a bit pattern of 7 flags, where 1 means the feature is on, 0 - off.

	bit	flag
x1xxxxxx	0	bold
	1	strike out
	2	inverse
	3	underline
	4	superscript
	5	subscript
	6	(always on)
	7	double underline

The inverse and underline attributes are also implemented in the ESC-G sequence. They are included for compatibility with earlier versions of the display driver.

Two pairs of enhancements are mutually exclusive:

1. superscript and subscript.
2. underline and double underline.

If both flags of a pair are set, the flag with the lower bit number takes precedence. The order of checking flags and applying enhancements to the character is as follows:

1) super/subscript
2) bold
3) strikeout
4) double/underline
5) inverse

The algorithms used for displaying the enhancements are as follows:

BOLD - character is OR'ed over itself one dot position to the right in the character cell.

SUPERSCRIPT - character is shifted up two dots and the top three rows of the character cell are ORed together to make room for the superscripted character.

SUBSCRIPT - character is shifted down two dots and the bottom three rows of the character cell are ORed together to make room for

the subscripted character.

UNDERLINE - the bottom row of the character cell is filled with dots.

DOUBLE UNDERLINE - the bottom two rows of the character cell are filled with dots.

STRIKEOUT - the fifth row of the character cell (first row is row one) is filled with dots.

The character set enhancements will not work in any cell smaller than 7 by 11 dots. The CCOS default character set is 6 by 10. If use of the enhancements is planned, setting up a STARTUP.TEXT file to automatically load a larger character set is recommended.

Special display driver control

Some driver functions are not accessible by using the high level language I/O or the UNITWRITE function. These other functions are accessed by using the UNITSTATUS call. The format of the call is:

```
UnitStatus(DisplayUnitNo,Buffer,Func);
```

where DisplayUnitNo and Func are integers and Buffer is a parameter block containing the parameters in the order shown. For CCOS version 1.1, the display unit number is 36.

Function	Code	Parameters
Read cursor position	0	xposition,yposition: integer;
Create window	1	NewWindowRec: WndRcd;
Delete window	2	WindowRec: WndRcd;
Select window	3	WindowRec: WndRcd;
Clear window	4	WindowRec: WndRcd;
Get window status	5	homex,homey,width,height: integer;
WriteBytes	6	bytecount: integer; pBuff: pBytes;
ReadBytes	7	bytecount: integer; pBuff: pBytes;
Load CRT Table	8	see below

For example, to retrieve the current X and Y position (relative to the current window, of course) of the cursor:

```
Const
  DispUnit = 36;

Var
  CPos : Record
           XCo : Integer;
           YCo : Integer
         End;

Begin
  UnitStatus(DispUnit,CPos,0);
  WriteLn('Current cusor X was: ',CPos.XCo);
  WriteLn('                Y was: ',CPos.YCo)
```

Note: When declaring records for UnitStatus calls, each field MUST have its type declared seperately. If the variable CPos had been declared as:

```
Var
  CPos : Record
           XCo,
           YCo : Integer
         End;
```

the fields XCO and YCo would have been allocated in memory in reverse order, and the cursor co-ordinates would have been returned in the wrong order.

LOADABLE CRT TABLES

The display driver provides user-loadable translation tables. This means that the CRT function control codes can be changed to emulate other terminals at the driver level. The driver contains a default translation table which is in effect when the system is booted. You can switch to a different table by loading the new table into memory and passing its address to the CRT driver to make it the current table:

```
UnitStatus(CRTunitnumber,Table,8);
```

where Table is the new translation table. Since the second parameter in the UnitStatus procedure is a VAR parameter, the CRT driver receives the address of the new table. If this address is nil (0) the default table becomes current.

The BLDCRT utility program can be used to build new CRT function tables. The BLDCRT utility is described in chapter 6.

The ASCII value of the control character is used as an index into the table. Non-negative bytes in the CRT table correspond to entries in the jump table of the CRT driver. Table indices from 0..$1F (first 2 rows) refer to control characters; table indices from $20..$7F refer to the character after an ESC. A byte value of $FF means the code is invalid or not implemented. This is the default table contained in the CRT driver:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	FF	FF	FF	FF	FF	FF	FF	08	03	06	01	00	02	05	FF	FF
10	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
20	FF	FF	FF	26	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
30	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	24	FF	FF
40	FF	00	01	02	03	0E	FF	11	04	FF	09	0A	14	15	FF	2C
50	26	0C	0F	FF	FF	FF	FF	0D	FF	0B	2E	FF	FF	FF	FF	FF
60	FF	16	19	18	05	2D	29	1A	FF	07	FF	FF	28	2A	1D	2B
70	27	1E	1F	1C	1B	20	21	22	23	17	10	FF	FF	FF	FF	FF

Each byte in the CRT table refers to a driver function. Here is the list of available functions (codes are in hex). The default sequence follows.

Hex	Function description	Default sequence(s)
00	cursor up	ESC A or CTL-K
01	cursor down (linefeed)	ESC B
02	cursor right	ESC C or CTL-L
03	cursor left (backspace)	ESC D or CTl-H
04	cursor home	ESC H
05	carriage return	CTL-M
06	tab	CTL-I
07	back tab	ESC i
08	sound bell	CTL-G
09	clear screen and home cursor	ESC J
0A	clear to end of line	ESC K
0B	clear to end of screen	ESC Y
0C	insert character	ESC Q
0D	delete character	ESC W
0E	insert line	ESC E
0F	delete line	ESC R
10	invert screen	ESC z
11	video command	ESC G mode
12	inverse video	
13	normal video	
14	set auto LF on CR	ESC L
15	suppress auto LF	ESC M
16	page mode on	ESC a
17	page mode off	ESC y
18	cursor on	ESC c
19	cursor off	ESC b
1A	graphics mode	ESC g
1B	text mode	ESC t
1C	scrolling on	ESC s
1D	scrolling off	ESC n
1E	insert mode on	ESC q
1F	insert mode off	ESC r
20	underscore cursor	ESC u
21	inverse cursor	ESC v
22	wrap at eoln	ESC w
23	no wrap at eoln	ESC x
24	CCOS type gotoxy	ESC = col row
25	CP/M type gotoxy	
26	skip two characters	ESC # c1 c2 & ESC P c1 c2
27	plot point	ESC p <parms>
28	draw line	ESC l <parms>
29	fill block	ESC f <parms>
2A	copy block	ESC m <parms>
2B	set origin	ESC o <parms>
2C	overstrike 2 chars	ESC O ch1 ch2
2D	character set enhancements	ESC e flags
2E	invert characters at cursor	ESC Z count

The DataComm Driver

The datacomm driver controls I/O at the two RS-232 ports on the Concept. The driver is interrupt-driven and will send and receive characters without explicit program control. Input is always accepted (until the input buffer is filled), and the contents of the output buffer are automatically sent. Once the data has been put in the buffer, the program need no longer be concerned with it.

This can lead to problems in some cases. Since the driver will always accept incoming characters (unless the interrupt mask is set to disable the interrupt), applications that receive data from the RS-232 ports may get a burst of garbage when first executed. Clearing the buffer with the UnitClear procedure avoids this, but since this also resets all the datacomm parameters (baud, parity, etc.) to their defaults, the settings should be read, saved, and restored after this operation.

Both input and output buffers are 256 bytes initially, but may be set as large as 32K bytes of necessary. In addition, alternate buffers of up to 32K may be set up for both input and output.

UnitRead

UnitRead reads data from the input buffer. If no data is present, the call will "hang" and wait for data to be received. To avoid this, use the UnitBusy function to check for the presence characters in the input buffer prior to issuing the UnitRead.

UnitWrite

UnitWrite write data to the output buffer. Note that the data will not necessarily be transmitted at the time it's written, since in most cases UnitWrite can fill the buffer much faster than the data can be transmitted.

UnitBusy

The UnitBusy function returns TRUE if there are characters in the input buffer.

UnitClear

Clears both the input and output buffers, as well as rsetting the datacomm parameters to the defaults. The default settings are:

Port 1: 1200 baud, space parity, 7 bits, XOn/XOff
Port 2: 9600 baud, no parity, 8 bits, XOn/XOff

UnitInstall, UnitMount, UnitUnmount

These unit procedure have no function in the datacomm driver. No errors will result from their use, but no action will occur.

In additional to the normal UNIT commands, the datacomm driver supports a number of special UnitStatus calls. These calls are described below. The datacomm driver supports three units: 0 for the printer (assumed to be connected to RS-232 port 2), 1 for dtacom1, and 2 for dtacom2.

Several of the datacomm parameters, such as baud rate and character size, are represented by single digit codes. These codes are used to refer to the specific settings and must be used in the following UnitStatus calls.

Baud rate codes

```
Baud300  = 0
Baud600  = 1
Baud1200 = 2
Baud2400 = 3
Baud4800 = 4
Baud9600 = 5
Baud19200= 6
```

Parity Codes

```
Parity disabled = 0
Parity odd      = 1
Parity even     = 2
Parity mark     = 3
Parity space    = 4
```

Port Codes

```
Port 1  = 0
Port 2  = 1
```

Character Size Codes

8 bits = 0
7 bits = 1

Handshake Codes

CTS line inverted	= 0
CTS line normal	= 1
DSR line inverted	= 2
DSR line normal	= 3
DCD line inverted	= 4
DCD line normal	= 5
XOn/XOff	= 6
Enq/Ack	= 7
Etx/Ack	= 8
No protocol	= 9

UnitStatus codes

Following are the special UnitStatus codes, their functions, and the required parameters.

Function code: 0
Name : Return write buffer free space
Param : Integer

Returns the number of byte available in the output (write) buffer. This call can be used to make sure that an application does not overflow the output buffer by writing data faster than it can be sent.

Function code: 1
Name : Return baud rate setting for unit
Param : Integer

Returns the baud rate code corresponding to the baud rate set for the device.

Function code: 2
Name : Return parity setting for unit
Param : Integer

Returns parity code corresponding to the parity setting for the unit.

```
Function code: 3
Name         : Set print unit / Return free space in input
               buffer
Param        : Integer
```

For unit 0 (the printer): Associates the /printer device with either RS-232 port 1 or 2.

For units 1 and 2 (the dtacoms): Returns the number of free bytes in the read (input) buffer.

```
Function code: 4
Name         : Return char size setting
Param        : Integer
```

Returns a 1 (7 bits) or 0 (8 bits) depending upon the character size setting for the specified unit.

```
Function code: 5
Name         : Return handshake type
Param        : Integer
```

Returns the handshake code for the setting used by the specified unit.

```
Function code: 6
Name         : Return datacomm unit settings
Param        : Record
                 BaudRate,
                 Parity,
                 Port,
                 CharSize,
                 HandShake : Integer
               End;
```

This call returns the parameters for the printer, dtacom1, or dtacom2. For example:

```
Var
  Status : Record
             Bd,
             Prty,
             Prt,
             ChSz,
             HndSk : Integer
           End;
```

```
Begin
  UnitStatus(1,Status,7);
```

```
Function code: 7
Name         : Return write buffer status
Param        : Record
                 BufferSize : Integer;
                 FreeSpace  : Integer;
                 HiWater    : Integer;
                 LoWater    : Integer;
                 InputDsAbld: Boolean;
                 OutptDsAbld: Boolean;
                 AutoLinFeed: Boolean;
                 AltBufAvail: Boolean;
                 AltBufAddr : pByte;
                 AltBufSize : Integer
               End;
```

Returns the write buffer status for the desired unit. The individual fields in the parameter block are:

BufferSize- The size of the current buffer, in bytes.

FreeSpace- The number of unused bytes in the buffer.

HiWater- ?

LoWater- ?

InputDsAbld- If TRUE, characters written to the unit will be ignored.

OutptDsAbld- If TRUE, characters in the write buffer will not be transmitted.

AutoLinFeed- If TRUE, a line feed will be sent after every carriage return.

AltBufAvail- TRUE if an alternate buffer is available.

AltBufAddr- The address of the alternate buffer.

AltBufSize- The size in bytes of the alternate buffer.

Function code: 8

```
Name          : Return read buffer status
Param         : Record
                  BufferSize : Integer;
                  FreeSpace  : Integer;
                  HiWater    : Integer;
                  LoWater    : Integer;
                  InputDsAbld: Boolean;
                  OutptDsAbld: Boolean;
                  LostData   : Boolean;
                  AltBufAvail: Boolean;
                  AltBufAddr : pByte;
                  AltBufSize : Integer
                End;
```

All parameters have the same meaning as the equivalent write buffer status parameters except:

LostData- If TRUE, parity, framing, or overrun errors, or buffer overflow, has resulted in the loss of some input data.

HiWater- The "high water mark." When this many characters are in the buffer, the driver will tell the host to stop sending characters using the assigned protocol. This prevents buffer overflow.

LoWater- The "low water mark." When the level of characters in the buffer has dropped to this point, the driver tells the host to resume transmission.

InputDsAbld- If TRUE, characters received by the unit will be ignored.

OutptDsAbld- If TRUE, characters in the write buffer cannot be read by the user.

```
Function code: 9
Name          : Set high water mark for read buffer
Param         : Integer
```

Sets the number of input characters for the high water mark. If the buffer fills to this point, the driver will force the host to cease transmission using the appropriate protocol.

Function code: 10
Name : Set low water mark for read buffer
Param : Integer

If reception of new data has been suspended by the driver due to a "high water" condition occuring, when the number of characters in the buffer drops to this level, the driver will allow the host to resume transmission.

Function code: 11
Name : Toggle read buffer output disable
Param : Nil

If disabled, the user will be unable to get characters from the buffer. The current status of this toggle can be obtained with UnitStatus call 7.

Function code: 12
Name : Toggle read buffer input disable
Param : Nil

If disabled, characters coming into the port will be dropped, and not put into the buffer.

Function code: 13
Name : Toggle write buffer output disable
Param : Nil

If disabled, characters in the output buffer will not be sent. The status of this flag can be checked with UnitStatus code 8.

Function code: 14
Name : Toggle write buffer input disable
Param : Nil

If disabled, characters written to the device will be thrown away and not transmitted.

Function code: 15
Name : Return number of characters in input buffer
Param : Integer

Returns the number of characters available in the input buffer. Useful in situations where no handshaking protocol can be used as it allows the application program to manage

the buffer.

Function code: 16
Name : Return number of characters in output buffer
Param : Integer

Returns number of characters in the output buffer.

Function code: 17
Name : Toggle auto linefeed
Param : Nil

Toggles the status of the auto linefeed flag. If set, line feeds (ASCII $0A characters) will be automatically sent after each carriage return (ASCII $0C). The status of this flag can be checked with UnitStatus code 8.

Function code: 18
Name : Set # of chars between Enq or Etx
Param : Integer

Only valid when the appropriate protocol is used. Sets # of characters that must be received after an Enq or Etx before Ack is sent.

Function code: 19
Name : Set alternate read buffer
Params : Longint, integer

The parameter to this call takes the form of a record:

```
Record
   NewBuf: LongInt;
   Size  : Integer
   End;
```

NewBuf contains the address of the alternate buffer, while Size contains its size in bytes. The maximum size is 32,767 bytes. After executing this function, all subsequent incoming data will be deposited in the alternate buffer. Calling this function with NewBuf set to 0 will cause the datacomm driver to reset to the primary input buffer.

Function code: 20
Name : Set alternate write buffer

Params : Longint, integer

The parameters and actions of this function are the same as those for function code 19, except that the buffer affected is the write buffer.

Function code: 21
Name : Clear read buffer
Param : Nil

Clears the read (input) buffer. If UnitBusy was true prior to the execution of this function, it will be false after execution. Attaches input to alternate read buffer if one has been established.

The Timer Driver

The CCOS timer driver performs three functions:

1. A table of user timer service routines is maintained and the various routines are serviced (if necessary) at each 50 millisecond timer interrupt.

2. VIA timer number 2 is used to produce the various bell sounds through the Concept speaker.

3. It maintains the real-time clock.

User service routines

Users can set up small service routines that the timer driver will automatically call (approximately) every 50 ms. This is NOT a precise time; and applications requiring a different or more precise interval must set up and monitor VIA timer #2. Timer #1 is for the exclusive use of the timer driver.

The UNITSTATUS procedure for the timer driver performs four functions related to the creation and maintenance of a table of user routines. These functions are:

UnitStatus function code	Function
1	Create a table entry
2	Delete a table entry
3	Enable a table entry
4	Disable a table entry

All the actions will place error condition codes within the IORESULT field in SYSCOM. Only two error conditions are defined. They are a) table full and b) no such entry (or invalid table entry ID).

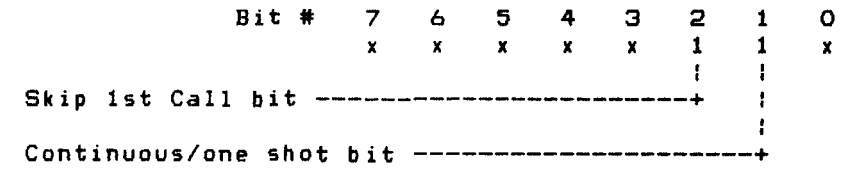
Function 1: Create a User Table Entry

This function takes as input the address of the user routine, the number of 50ms timeouts before service (range of 1 to 65,535), and a word of flags. It returns the table entry ID (word) of the entry made.

Only two bits are currently used in the flags word. Bit one is the continuous/one shot flag. When it is clear, the table entry is used (every COUNT timeouts) until is is

removed by the Delete call. When it is set, the associated routine is deleted after the first time it is called.

Bit 2, when set, serves as a "skip first call" flag. This flag causes the timer driver to ignore the FIRST call to be made to the corresponding user routine. All subsequent call are performed. The need for this flag arises due to the fact that the Create operation is performed asynchronously to the interrupts generated by the driver, which implies that the first call to the user routine will be made before the specified number of 50ms timeouts. This flag is set when a desired MINIMUM time is necessary before a user routine is called.



Below is a Pascal program fragment that creates a timer parameter block:

```
CONST
        OneShot    = 2;
        SkipFirst  = 4;
        Timer      = 34;
        Create     = 1;       { UnitStatus function code }

TYPE
  TimerRec :
       record
          UserServiceRtn : pointer; {Input parameter}
          Count          : integer; {Input parameter}
          Flags          : integer; {Input parameter}
          TableID        : integer  {Output parameter}
        end;

Var
  TimerDemo : TimerRec;

Begin
  With TimerDemo Do Begin
    UserServiceRtn := Pointer(@(<some user stuff>));
    Count := 10;        { Every 500ms, more or less }
```

```
    Flags := 0;
    TableID := 0;       { Set by UnitStatus }
    End;
  UnitStatus(Timer,TimerDemo,Create); { Create entry }
```

Function 2: Delete a user table entry

The delete table entry procedure takes the table entry ID (word) for the entry desired and deletes that entry from the table. This table entry ID is the same value returned by the Create function. The Delete procedure has 1 input parameter and no (zero) output parameters, for a total of 2 bytes.

```
UnitStatus(Timer,TableID,2);
```

In this example, TableID is an integer containing the ID of the user routine (returned by Create) to delete.

Function 3: Disable user routine

The disable procedure allows the user to prevent the Timer interrupt routine from calling the user's timer service routine without deleting the user's table entry. The procedure has one input parameter, the table entry ID of the entry to disable. It has no output parameters. The total number of bytes for this procedure's parameter block is 2.

Example:

```
UnitStatus(Timer,TableID,3);
```

Function 4: Enable user routine

The Enable procedure allows the user to restart a table entry. It clears the Disable condition and resets the down counter to its starting condition. It can be used for either starting up a Disabled table entry or restarting a running entry before it causes a call to the service routine. This procedure has only 1 parameter, the table entry ID, for a total of 2 bytes.

Example:

```
UnitStatus(Timer,TableID,4);
```

Notes on user routines

User timer service routines are called during the interrupt processing. They should take a minimal amount of time; 500 microseconds is the recommended maximum. Therefore, don't use them for doing any significant processing. They would be best used for setting flags informing the program that a timeout event has occurred.

The table operations are not indivisible. Therefore, a user timer service routine may be called while the user is making a delete or disable call. Consequently, any operation done by the user service routine must be easily reversable after the table operation call. This only reinforces the need for keeping the user service routines simple with few destructive operations.

All user timer service routines must return to the Timer driver interrupt routine via the RTS instruction. Also, no argument passing is available to the the user's timer service routine.

For Pascal programs which must incorporate user timer interrupt service routine there are a few restrictions. Obviously, the routines must be coded in machine language and linked as externals to the pascal program. Only procedures can be used, since the user routines cannot return results. The procedure must be global. The procedure must return to the Timer driver and cannot use Global GOTO's. When the table entry is created, the routine that calls the timer driver create function must be a resident part of the program which contains the user's service routine procedure.

The Bell service

The timer driver's Bell service is also activated by a UNITSTATUS call. The code is 0, and the parameter passed is the Bell parameter block. The declaration for the Bell parameter block is:

```
Type
  BellParm: Record
              Freq : Integer;
              Ptrn : Byte;
              Filr : Byte;
              Drtn : Integer
            End;
```

The first field is the frequency of the tone to be produced. The second field contains a pattern of 1s and 0s which

describe the way in which the speaker will be toggled; it can be used to produce interesting variations in the note produced. The Filr field is ignored, and is included as a filler to align the next parameter on a byte boundary. The last field is the duration of the tone produced, in terms of 50ms timeouts.

Real time clock service

The third service, the time and date clock, is accessed via the Unitread and Unitwrite driver interfaces. The clock interfaces require a Clock Parameter Block. This is a fixed length, fixed format interface. Unlike other drivers, the timer driver cannot do reads and writes of variable length data strings. The Clock Parameter Block is one field longer for the Unitwrite procedure then for the Unitread procedure. The following is a Pascal fragment which describes the Clock Parameter Block.

```
type
   clockPB = record
               DayofWeek : Integer;   {1..7}
               Month     : Integer;   {1..12}
               Day       : Integer;   {1..31}
               Hour      : Integer;   {0..23}
               Mins      : Integer;   {0..59}
               Secs      : Integer;   {0..59}
               Tenths    : Integer;   {0..9}
               case INTEGER of
                 0 : ( {Write only} LeapYear : 0..3);
                end {variant}
             end;{record}
```

Though each field is a sub range of the integer type and could fit in less space then a word, each field MUST be a word long. Therefore, declare each field as an integer, but restrict your useage to the valid range. The real parameter block is 8 bytes long.

The driver does validate the range in case of a user error. Unfortunately, the Day parameter is not completely verified. If the month specified is February (Month := 2), but the Day is 30, the driver does not report an error.

Except for the fields DayofWeek, Tenths, and LeapYear the field meanings are self-explanatory. DayofWeek specifies which day it is, such as Monday, Tuesday, etc. The value 1 specifies Sunday while 7 specifies Saturday. The field Tenths is tenths of a second. Finally, the clock has no register set for years. However, to keep the correct date

after a year change the clock tracks if this year is a leap year. The LeapYear field specifies the number of years AFTER the last leap year. Therefore, if this year is leap year then the field should be assigned to 0 (zero). Consequently, if last year was leap year then the field should be assigned to 1 (one), and so forth. The LeapYear field is only used when setting the clock (Unitwrite). It is not returned when reading the clock (Unitread).

To set the clock, first fill in all the fields in the clock parameter Block, including the LeapYear field. Next, call the timer driver Unitwrite to set the clock.

```
Unitwrite( TimerDriverNumber, WriteParmBlock, 8);
```

The driver will set the IORESULT variable if the parameter block is in error.

To read the clock just call the timer driver Unitread with the Clock Parameter Block.

```
Unitread( TimerDriverNumber, ReadParmBlock, 8);
```

Unitinstall

The Timer driver unitinstall procedure sets-up the VIA, initializes the timer table to no entries value, starts the 50 ms. interrupt driven interval timer (VIA timer #1), and initializes the clock chip. Clock initialization makes sure the clock is running, the chip is not in test mode, and the interrupt capabilities of the chip are reset. The chip can interrupt, however, it is not connected to the system interrupt system and it's interrupt is ignored. The interrupt is cleared and reset so it does not effect the running of the clock.

Unitunmount

The Timer driver unitunmount procedure turns off the VIA timer interrupt for Timer #2. It also installs in the timer interrupt vector a pointer to a ReTurn from Exception (RTE) instruction to insure system integrity in case of a spurious level 5 interrupt.

The Disk Drivers

There are three basic types of disks that the Concept can use:

1. A hard disk;
2. An 8 inch floppy disk;
3. An Apple floppy disk.

With the exception of the Apple floppy disk driver, which is read-only, all of these drivers are functionally identical. A program need not be concerned whether the disk on unit 12 is a hard disk or a floppy- the UNIT I/O procedures all operate in the same fashion.

The two possible differences in the various type of disks are differences in capacities and UNITSTATUS calls.

The disk system is not interrupt driven, so the UNITCLEAR and UNITBUSY procedures have no function. UNITREAD and UNITWRITE may be used to directly read and write blocks from the disks.

The Revision B Floppy Disk Driver

8 inch floppy formats

The driver can handle most variations of the standard single and double density formats. The standard single density format is the IBM 3740 format. The standard double density format is the "System 34" format. For single density it can use 128, 256 or 512 byte sectors and any number of sectors per track, the standard is 26 128 byte sectors per track. For double density it can use 256 or 512 byte sectors and any number of sectors per track, the standard is 26 256 byte sectors per track. It can use any number less than 128 as the number of tracks per side. The standard is 77 for both density. It also can use single and double sided diskettes in either format.

A double sided diskette can be distinguished from a single sided diskette by the location of the index on the diskette Single sided diskettes have index holes in the center of the diskette, while double sided diskette have the index hole off-center.

When the driver finds that a new diskette has been placed in the drive it will determine the density, number of sides used and the sector size any action from the user. The

driver can also determine the number sectors per track base on the sector size if the user is using standard diskettes.

The standard sectors per track are :

Density	Bytes per sector	sectors per track
Single	128	26
Single	256	15
Single	512	8
Double	256	26
Double	512	15

Logical interleave, skew, and track offset

Interleave is the logical reordering of physical sector numbers on a single track. Skew is the varying physical sector number used as the first logical sector on the track. Track offset is the physical track number, from 0 to number of tracks per side, used as logical track 0.

System Interface and driver operation

The driver uses slot static RAM for it's static local tables. Each slot has own page(256 bytes of RAM) but drives in the same slot have the same tables, i.e. interleave, skew, etc. The driver allocates its local temporary data area on the stack.

Unitinstall

The install procedure must be called at least twice for every controller card in the system. For the first call, the drive number in the device table for the unit must be set to zero(0) and the slot number field set to the correct slot number. The unitinstall procedure will then determine the number of drives connected to the controller card. Drives addresses range from 0 to 3. The driver starts searching at drive address 0 and looks to find the first drive number that does not respond. It reports the number of drives found in the number of drives field of the slot array in the system slot table

The unitinstall procedure is then called with a different unit number for each drive to be mounted. The calling program must fill in the slot number and drive number field in the unit's device table entry before calling the install

procedure. Legal drive numbers are 1-4.

One copy of the driver in the system can be used for all the floppy units. The driver keeps the static information on each controller in the slot's RAM page in the static RAM. Therefore, multiple drives attached to different controller cards can have different parameters.

Unitunmount

Does nothing.

Unitread

Reads blocks from floppy. It reads single/double density and single/double sided. Reads partial blocks. Returns to user only the number of bytes requested starting at the START BLOCK parameter passed to driver. Partial blocks start at beginning of block, not in middle of block.

Unitwrite

Writes blocks from floppy. It writes single/double density and single/double sided. Partial blocks can be written, but only from the start of a block; i.e. the middle 128 bytes of a block could not be seperately written.

Unitbusy

Always true.

Unitclear

Initializes the floppy disk controller then restores drive to track zero and determines type of floppy in drive.

Unitstatus

The revision B floppy driver supports a number of new UNIT-STATUS calls. These are listed below.

1) change logical sector interleave

function code : 0

```
        Parameter Block :
        Interleave = word, range = 1..(sectors per track).

2) change logical skew

        function code :  1
        Parameter Block :
        Skew = word, range = 1..(sectors per track).

3) change track offset

        function code :  2
        Parameter Block :
        StartTrack = word, range = 0..(tracks per side).

4) change step rate

        function code :  3
        Parameter Block :
        StepRate = word, range = 1..16.

5) change number of sectors per track

        function code :  4
        Parameter Block :
        Sectors per track = word, range = 0..127.

A value of 0 means use the default internal table
based on density and sector size.

6) change number of tracks per side

        function code :  5
        Parameter Block :
        Tracks per side = word, range = 0..127.

A value of 0 means use the default 48 TPI standard
number of tracks, 77.

7) change timeout counter for waiting for drive to come
   Ready.

        function code :  6
        Parameter Block :
        TimeOut = longint, range = positive integer.

8) get driver state.  the function returns :

        all changeable attributes

        a) interleave
```

```
b) skew
c) track offset
d) step rate
e) sectors per track
f) tracks per side
g) Ready wait timeout

and driver determined diskette states
(queries floppy to find out)

a) presence of a floppy in drive
b) density
c) sides
d) sector size

function code :  7

Parameter Block :  all fields set by driver.

record
intrlv,              {interleave}
skew,
StartTrack,
StepRate,
spt,                 {sectors per track}
tps,                 {tracks per side}
SectorSize : integer;
TimeOut : longint;
Diskette,            {true if diskette in drive}
OneSided,            {true if 1 side only}
SnglDensity,         {true if single density}
UserSPT : boolean;   {true if user set spt}
end;

The Diskette field must be true to have valid One-
Side, SectorSize and SnglDensity fields' values.
```

IOresult codes returned by driver

Name	Value	Comment
IOEcrcer	1	CRC error
IOEioreq	3	Invalid I/O request
IOEnebhrd	4	Nebulous hardware error
IOEoffln	5	Drive off line
IOEwrprot	16	Device write protected
IOEinvblk	18	Invalid block number
IOEflpto	24	Timeout error
IOEnoTO	25	Cannot restore to track 0
IOEnfmtd	26	Disk not formatted

IOEinvsct	27	Invalid sector length error
IOEwrngC	28	Read wrong track
IOEbdtrk	29	Track marked as bad (IBM spec
IOEuiopm	54	Invalid unit I/O parameter
IOEfnccd	56	Invalid function code

The Enhanced Printer Driver

The enhanced printed driver operates as a "front end" pre-processor to the standard /printer or /dtacom driver. It supports the printing of the on-screen (display driver) enhancements of underlining, superscript, subscript, strike out, and boldface (assuming the printer can print them).

The driver operates by intercepting the ESCAPE e enhancement sequence and translating it to the sequence of characters required by the printer in use. The translation is directed by a printer action table (created with the BLDACT utility supplied by Corvus). In addition, a seperate table of alter nate characters allows the user to define and print special characters, which are entered through the keyboard by hold-ing the [ALT] key down while pressing another key. The user can define special characters for these ASCII codes (128-255) and direct their printing with the appropriately con-structed alternate character table. The alternate character table is created with the supplied BLDALT utility

The enhanced printer driver normally operates through data-comm port 2 and supports all normal datacomm parameters

The printer driver uses the following standard driver com-mands :

1) Unitwrite : send characters to printer or turn on/off enhancements.

2) Unitbusy : Boolean response stating whether or not the printer driver is able to accept more charac ters.

3) Unitclear : Clear the internal buffer of the driver and reinitialize the state of the driver enhance ments.

4) Unitinstall : Initializes the driver to its default state.

   a) Attach the driver to the default Datacom driver,
   b) initialize the UART to its default state,
   c) initialize the driver's internal variables,

5) Unitunmount : Deattach driver from system.

6) Unitstatus : used for several driver dependent functions

   a) change or return the UART state
   b) return state of driver, i.e. buffers.

c) change state of driver Unitwrite to Transparent mode or Translate mode.
d) install a new Alternate Character Translation Table.
e) Attach driver to another unit.
f) select the pitch the driver/printer is working in
g) select the number of lines per inch.
h) install a new Printer Action table
i) return state of pitch and number of lines per inch.

Unitwrite

The Unitwrite function sends characters to the printer.

When the driver is in Translation mode, it will translate the character stream going to the printer. In this mode the driver can add or remove enhancements to the characters sent to the printer. Enhancements are controlled by an escape character sequence. The sequence is :

ESCAPE e FLAGBYTE

where ESCAPE is the control character escape (hex value $1B), e is the ASCII character for lower case e (hex value $65), and FLAGBYTE is a byte (one character) of bit flags representing the state of the enhancement flags. The FLAGBYTE has the form :

Bit Number	Function
0	Bold enhancement
1	Strike Out enhancement
2	Inverse enhancement (not implemented)
3	Underline enhancement
4	SuperScript enhancement
5	SubScript enhancement
6	bit must always be 1
7	Double Underline enhancement (not implemented)

The Inverse and Double Underline enhancements are not implemented in the printer driver. Bit 6 is always 1 so the byte can never have a value in the control code range (hex $00 through $1F). This is necessary to prevent trouble when transmitting this character sequence over DataCom lines.

After an printer enhancement sequence is started, all subsequent characters will be printed with the specified enhancements until a terminating command (enhancement sequence with

all flag bits off) is sent.

The strikeout enhancement is performed on a per character basis. A character to be struckout is first printed. A backspace operation is performed on the printer. Finally, the overstrike character, the minus sign "-" (hex value $2D), is printed. Any printer used with this driver must be able to perform a single character back space using the backspace character code (hex value $08). Furthermore, the printer must perform the backspace operation using the same character spacing used on the character that is struckout. This last requirement is needed to properly do proportional spacing.

Superscript and subscript are performed by first making the forms advance distance shorter. Then a line feed or reverse line feed operation is done to lower or raise the print position to an area between lines. The forms advance distance is then restored to the normal value for the current number of lines per inch. This is necessary in case the enhancement is carried on after a CR. When the enhancement is turned off, the same operations are performed except the opposite forms advance direction is used.

The function in translate mode accepts another escape sequence to do overstrike. This sequence is :

ESCAPE O <FirstChat> <OverChar>

where ESCAPE is the control character escape (hex value $1B), O is the ASCII character for upper case o (hex value $4F), FirstChar is the character to be overstruck, and OverChar is the character to overstrike FirstChar.

The overstrike enhancement is performed with the same method as the strikeout enhancement. It also has the same requirements in terms of the backspace operation on the printer.

The Unitwrite function will also perform proportional spacing on a line per line basis. Proportional spacing does not have to start at the beginning of the line or stop at the end of the line- it can begin and end anywhere. To turn on proportional spacing, send the following escape character sequence to the driver before sending all the characters to be proportionally spaced.

ESCAPE P <# of Chars> <# of Pads>

where ESCAPE is the ASCII escape char ($1B), P is the proportional spacing indicator character upper case P (hex value $50), <# of_Chars> is an unsigned byte representing

the number of characters to do the proportional spacing with, and >## of Pads> is an unsigned byte representing the number of character spaces to pad with.

The driver performs proportional spacing by adding extra microspaces between each of the character symbols. This mechanism takes the number of full character pads needed an divides the space as equally as possible among the characters. It will add an extra 1, 2, or 3 1/120ths of an inch between each character. The maximum number of character space pads is a function of the pitch.

10 pitch : 3/12 * <# of_Chars> maximum number of pads
12 pitch : 3/10 * <# of Chars> maximum number of pads

If the number of pads needed is greater than the amount of microspaces the driver can add the driver will return an error I/O result code. Unlike with most errors the driver detects, it does not immediately return to the caller if more characters are available. It processes the remaining characters with the proportional spacing off.

The driver recognizes a special character, hex code $A0, for alternate space. This is the "rubber" space code. It is used by some applications such as EdWord for proportional spacing. This character is not sent to the printer, but is used by the driver. If proportional spacing is turned off, then $A0 is treated as an alternate character and is used to index into the alternate character translation table.

All characters, except characters within the escape sequences and the "rubber" space previously mentioned, which are in the alternate character set (hex values $A0 to $FF) are translated via the Alternate Character Translation Table into a character sequence which will get the character's symbol printed. This translation is performed prior to any other processing of the Unitwrite character stream. Therefore, the escape sequence for overstrike may be used as the translated sequence for an alternate character. The table has the following form :

```
RECORD
   PTR_CHAR_SEQ : pCHAR_SEQ;
   CHAR_INDEX   : ARRAY[0..96] OF Unsigned_byte;
   CHAR_SEQ     : ARRAY[0..LAST_STR] OF str8;
   END;
```

PTR_CHAR_SEQ is a pointer to the CHAR_SEQ array. The array CHAR_INDEX is indexed by the alternate character minus $A0. The value is either an index into the array CHAR_SEQ or $FF

which means no CHAR_SEQ string exists for this alternate character. CHARSEQ is a variable length character sequence (with a maximum length of 8 characters) which is sent to the printer for any alternate character with a CHAR_INDEX which points to it. Therefore, several alternate character codes can use a single character sequence. Alternate characters with a CHAR_INDEX value of $FF do not have a character sequence- they are sent to the printer with bit 7 clear, and translated to a 7 bit character code.

The driver has a default table defined for the NEC 7710 printer with NEC thimble Courier 72.

If the driver is in transparent mode NONE of the previously mentioned character stream processing is performed. The characters are sent directly to the printer with no enhancements. This mode is useful for sending special printer control escape sequences directly to the printer. It is also useful for utilizing the graphics capabilities of some printers.

Unitbusy

This function states whether the printer driver can accept characters into it's buffer. TRUE means the buffer is NOT

Unitstatus

All functions that are passed through to the attached unit have function codes less than $80. All internally processed functions, except the port select function, have function codes greater than or equal to $80 and less than or equal to $FF. The select port function is processed internally but has a function code less than $80 in order to maintain compatibility with the old printer driver.

The general form of the UnitStatus call to the enhanced printer driver is:

```
Unitstatus( PrinterUnitNumber, ParamBlock, FuncCode );
```

where PrinterUnitNumber is the unit number for the printer driver (6), ParamBlock is the parameter block which varies according to the function performed, and FuncCode is a word containing the function code.

There are several classes of UnitStatus calls:

1) pass through functions (compatible with old driver)

Baud rates	Function code 1
parity	Function code 2
word size	Function code 4
handshake protocol	Function code 5

2) internally processed functions (compatible with old driver)

port	Function code 3

b) return state of driver, i.e. buffers.

Free space in transmit buffer Function code 0

The Buffer should be an integer; the call will return the number of bytes free in the transmit buffer.

c) change state of driver Unitwrite to Transparent mode or Translate mode.

This is a very important call!

When the driver is in transparent mode, all enhancements performed by the printer are turned off. Thus, all characters will be sent directly to the printer with no processing performed. Since characters in the output buffer will be affected, the user should not switch driver modes unless the buffer is empty. The default state of the driver is the translate mode.

d) install a new Alternate Character Translation Table.

The function code for this call is $81; Buffer is a pointer to the address of the new translation table. The default is the table for the NEC 7710 type printer with the Courier 72 thimble.

If the pointer to the new table is Nil (equal to zero) then the default Alternate Character Translation table is used.

e) Attach driver to another unit.

The function allows the user to send the output to any driver mounted in the system, using the UnitWrite and Unit-

Status interfaces. Appendix B contains the listing of a Pascal program that will attach a new output driver to the enhanced printer driver.

The function code for this call is $82; Buffer in an integer which contains the unit number of the new driver.

f) Change the pitch (CPI) of the driver.

The function code for this call is $83; Buffer is an integer contains either a 0 (12 pitch) or a 1 (10 pitch)

The default is 10 pitch.

g) Change the lines per inch of the driver.

The function code for this call is $84; Buffer is an integer which is either 0 (8 LPI) or 1 (6 LPI).

The default is 6 lines per inch.

h) Install a new Printer Action Table

The function code for this call is $85; Buffer points to the new printer action table.

If the pointer to the new table is Nil (equal to zero) then the default Action table is used.

The following is a list of the functions necessary for a given printer:

1) Turn on underline.
2) Turn off underline.
3) Turn on bold (Nec calls this enhancement Shadow).
3) Turn off bold
4) Reverse (or Negative) line feed. This function must be affected by a change to the form advance distance (see 5).
5) Change form advance distance for variable line feed size. This is used with 4 and 5 to perform and superscript.
6) Change the character spacing. This is needed for proportional spacing.

The table has the following structure, represented as a pseudo Pascal record. It is currently 102 bytes long.

```
record
     UnderLineOn     : str7;
     UnderLineOff    : str7;
     BoldOn          : str7;
     BoldOff         : str7;
     RevrsLF         : str7;
     BackSpace       : str7;
     SixLinesInch    : record
                         SubSuperFormAdv :   str7;
                         NormalFormAdv   :   str7;
                         end;
     EightLinesInch  : record
                         SubSuperFormAdv :   str7;
                         NormalFormAdv   :   str7;
                         end;
     Pitch10         : record
                         NormalSpacing   :   str7;
                         Micro1Extra     :   str7;
                         Micro2Extra     :   str7;
                         Micro3Extra     :   str7;
                         end;
     Pitch12         : record
                         NormalSpacing   :   str7;
                         Micro1Extra     :   str7;
                         Micro2Extra     :   str7;
                         Micro3Extra     :   str7;
                         end;
     end;
```

Str7 is a Pascal string with a maximum of 7 characters, not including the leading length byte.

Field descriptions:

UnderLineOn : Character sequence for printer to turn underline on. This must stay on until sent sequence to turn underline off.

UnderLineOff : Character sequence for printer to turn underline off.

BoldOn : Character sequence for printer to turn bold printing on. This enhancement is called Shadow on the NEC spinwriter. This enhancement is assumed to turn off automatically after the printer receives a CR (Carriage Return hex value $0D) character.

BoldOff : Character sequence for printer to turn bold

printing off.

RevrsLF : Character sequence for printer to perform a reverse linefeed operation. This function is necessary for subscript and superscript.

BackSpace : Character sequence for printer to perform a back space operation. On most printers this is the code $08 for BS. This function is necessary for overstrike and strikeout.

SixLinesInch : This record's fields are used when the printer is in 6 lines per inch mode.

EightLnsInch : This record's fields are used when the printer is in 8 lines per inch mode.

SubSuperFormAdv : Character sequence to change the form advance distance to roughly a quarter of the normal form advance distance. This field varies according to the lines per inch used by the printer. For the Nec 7710 printer using 6 lines per inch, this form advance distance is 2/48 of an inch. The character sequence is escape, "J", "Q". This function must change the form advance distance on the reverse line feed operation as well as the line feed operation.

NormalFormAdv : Character sequence to change the form advance distance to the standard distance used by the current pitch. This field varies according to the lines per inch used by the printer. For the Nec 7710 printer using 6 lines per inch, the normal form advance distance is 8/48 of an inch. The character sequence is escape, "J", "W".

Pitch10 : This record's fields are used when the printer is in 10 pitch or 10 characters per inch (CPI).

Pitch12 : This record's fields are used when the printer is in 12 pitch or 12 CPI.

NormalSpacing : Character sequence to set the printer spacing to normal distance for the given pitch. For the NEC 7710 running at 10 pitch the spacing value is 12/120 of an inch and the character sequence is escape, "J", "L".

Micro1Extra : Character sequence to change the character spacing by adding an extra 1/120th of an inch. For the NEC 7710 running at 10 pitch the spacing value is 13/120 of an inch and the character sequence is escape, "J", "M".

Micro2Extra : Character sequence to change the character spacing by adding an extra 2/120ths of an inch. For the NEC 7710 running at 10 pitch the spacing value is 14/120 of an inch and the character sequence is escape, "J", "N".

Micro3Extra : Character sequence to change the character spacing by adding an extra 3/120th of an inch. For the NEC 7710 running at 10 pitch the spacing value is 15/120 of an inch and the character sequence is escape, "J", "O".

Everything else not covered in chapters 1-5 | 6

This chapter deals with all of the other CCOS ancillary information and miscellanea not covered in the first four chapters. This includes startup and exec files, as well as the command line parameters accepted by the systems utilities.

Also covered are use of the linker, including the linking of Pascal, FORTRAN, and machine code segements; using the library utility to maintain custom routines in libraries, instructions on the use of the Pascal and FORTRAN compilers, and the use of various system utility programs.

Use of EXEC Files

Exec files are simple lists of commands in a standard text file that can be used to automate some processes. An example would be the automatic compilation and linkage of a Pascal or FORTRAN program.

Each record (line) in an Exec file is interpreted just as if it had been typed on the command line by the user. The format for a command line is:

(command) (arg 1) (arg 2) ... (arg n) (< ifile) (> ofile)

where the values of the command and arguments depend upon what's being done. The command parameter represents the system command or program file to be executed. A program file name with a ! prefix indicates that the program is in the system volume /CCSYS.

IFILE and OFILE are the input and output file for the program I/O. These optional parameters make use of the I/O redirection facility in CCOS.

Comments may be imbedded in a command file by using :, !, {, or } as the first character of the line. The special command "PAUSE" can be used and followed with a text message whose context indicates a Y or N reply. Y replies continue Exec file processing; "N" replies abort the process. When a PAUSE record is encountered, Exec file processing halts and the text message is displayed. The system accepts a keystroke from the user and evaluates as described.

CCOS recognizes a special command file with the name START-UP.TEXT if it is on volume 5. If CCOS finds this file during a boot, it will automatically be executed. This facility can be used to automatically configure a system (i.e. display character set, printer parameters, etc.) to a user's requirements.

Examples of Exec files are given after the system utility parameters.

File Manager Parameters

The file manager commands are as follows:

```
CATFIL File1 (File2...) > Outputfile
CPYFIL -VDstVol (-D) (-Q) (-S) SrcFil1 (SrcFil2...)
CRUNCH (-Q) /Vol1 (/Vol2...)
DELFIL (-Q) (-T) File1 (File2...)
```

```
FLPDIR /Volume
LSTFIL File1 (File2...)
LSTVOL (-B) (-H) (-L) /Vol1 (/Vol2...)
LSTVOL /                          (Lists default volume)
LSTVOL !                          (Lists system volume)
MAKFIL NewName1[length] (NewName2[Length]...)
RENFIL OldName NewName
RENFIL /OldVol /NewVol
SETVOL /Volume
```

All of these commands must be prefixed with !CC.FILMGR.

System Manager Parameters

The system manager commands are:

```
SETDAT (NewDate)
SETTIM (NewTime)
DRVVRS                            ( Display driver versions )
```

SETDAT and SETTIM will display the date or time if no parameters are given, or set the date or time if parameters are given. The date parameter must be in the form dd-mmm-yy, and the time parameter must be in the form HH:MM:SS (24 hour time).

These commands must be prefixed with !CC.SYSMGR.

Window Manager Parameters

The window manager parameters are:

```
BOXWND                  - Box or unbox current window.
CLRWND                  - Clear current window.
CSDISP Filename         - Load display char set for current
                          window.
CSKYBD Filename         - Load keyboard character set.
DEFSCN                  - Clear screen and display border.
DEFTTL                  - Update screen data.
REVBKG                  - Reverse window background.
SCRLMD                  - Toggle scroll mode.
```

All of these commands must be prefixed with !CC.WNDMGR. Note that you cannot create, delete, or select windows from a command file.

Parameters for DataComm and Printer

The SETDCP utility allows the user to set the protocol and parameters for the datacomm drivers. Command lines for setting up the datacomm and printer drivers take the form:

KeyWord=Parameter

1. NO BLANKS are permitted within a parameter

2. Parameters and keywords can be abbreviated to their shortest UNIQUE string.

Key Words	Parameters
UNIT	PRINTER, DC1, DC2
BAUD	300, 600, 1200, 2400, 4800, 9600, 19200
PARITY	DISABLED, EVEN, ODD, MARK, SPACE
HANDSHAKE	LINE/CTS/NORMAL, LINE/CTS/INVERTED, LINE/DSR/NORMAL, LINE/DSR/INVERTED, LINE/DCD/NORMAL, LINE/DCD/INVERTED, XONXOFF ENQACK ETXACK NONE
DATACOM	1, 2 (Only for printer)
CHARSIZE	7, 8
ALTCHARTABLE	<file name>
ACTIONTABLE	<file name>
CPI	10, 12
LPI	6, 8
AUTOLINEFEED	ON, OFF

Default values:

DataComm 1 (RS-232 port 1)

```
BAUD=9600
PARITY=DISABLED
HANDSHAKE=XONXOFF
CHARSIZE=8
AUTOLINEFEED=ON
```

DataComm 2 (RS-232 port 2)

```
BAUD=4800
PARITY=DISABLED
HANDSHAKE=LINE/DSR/NORMAL
CHARSIZE=8
AUTOLINEFEED=ON
```

Printer (configured for NEC 7700 Spinwriter)

```
BAUD=1200
PARITY=SPACE
HANDSHAKE=XONXOFF
CHARSIZE=7
DATACOM=2
AUTOLINEFEED=ON
CPI=10
LPI=6
```

Note: The UNIT parameter MUST be specified before any other parameter!

Parameters for spooler and despooler

For the spooler, command line parameters are as follows:

Keywords	Parameters
Set or Alternate	1,2,3,4,5
Include	<include string>
Message	<message string>
New	<new page string>
Text	Yes,No,True,False,Y,N,T,F
Enhance	Yes,No,True,False,Y,N,T,F
Pipe	<file name>

Defaults are:

```
Set = 5
Include = {$I
New =  PG
Text = YES
Message = Route to <user> on <date>
Enhance = YES       { TRUE strips enhancements }
Pipe = PRINTER
```

The Set and Alternate keywords specify that spooling be directed to a slot other than the default of 5. The Include and New keywords specify the strings that cause the spooler to recognize include files and page breaks. For example, EdWord uses PG to specify a page break, while the Pascal compiler uses {$P}. The Message keyword specifies a message that is printed on the title page of the spooled text file. The default message is "Route to <user name> on <date>."

Example: Spooling a Pascal program listing:

```
Spool N={$PG I={$I /Pascal/Newprog
```

Note that the new page and include file parameters are set correctly for a Pascal file. The spooler default for include file is the Pascal notation, while the default for the new page is EdWord notation. EdWord doesn't use include files, so the Include parameter can be ignored. When spooling a Pascal file, only one default need be changed.

Since the spooler can also be used to send non-textual (i.e. program and data files) to another machine, the Text flag is provided to allow the user to disable some of the translation that normally takes place on text files. Non-textual files should never be spooled to a printer!

Example: Spooling a data file

```
Spool T=N P=Fiscal /Data1/Fiscal83.DTA
```

Note that a new pipe name was specified to prevent the default pipe name of PRINTER from being used.

Notes: All keywords may be abbreviated to their first letter. If Text is false, the Include, New page, and Enhance parameters are invalid.

The despooler takes a seperate set of parameters from the spooler, keeping only the Set/Alternatre and Pipe parameters in common.

Keywords	Parameters
Device	<file name>
Expand	integer
Header	True, False, Yes, No, T, F, Y, N
Linefeed	True, False, Yes, No, T, F, Y, N
Maximum lines/page	integer
Trailer page	True, False, Yes, No, T, F, Y, N

Defaults are:

```
Device = PRINTER
Expand = 8
Header = YES
Linefeed = YES
Maximum = 58
Trailer = YES
```

The Device parameter specifies the device or file name to send the despooled output to. Expand takes as a parameter the number of spaces to expand tab characters. Header and Trailer specify the printing of header or trailer pages, useful in seperating files on a network printer. Maximum sets the maximum number of lines per page before a new page is forced. Linefeed forces a linefeed after every carriage return.

Example: Despooling the Pascal file:

```
Despool D=PRINTER
```

Example: Despooling the data file:

```
Despool D=/MyVol/Fiscal83
```

Examples

The following example is a hypothetical STARTUP.TEXT file:

```
{ First we'll set up our printer }
!CC.SETDCP U=PRINTER BAUD=1200 PAR=DISABLED HAND=XONXOFF
{ Now our modem }
!CC.SETDCP U=DC1 BAUD=300 CHAR=8 HAND=XONXOFF
{ Now load a big character set }
!CC.WNDMGR CSDISP /CCUTIL/CSD.09.14.ALT
{ Fire up EdWord }
!ED /WORK/NOVEL
```

The first two lines could have been combined and abbreviated as follows:

```
!CC.SETDCP U=P B=1200 P=D H=XONXOFF U=DC1 B=300 C=8 H=XONXOF
```

BLDACT- The printer action table utility

The BLDACT utility normally resides on the /CCUTIL volume and is used to build printer action tables. A printer action table consists of a list of the special character sequences used to enable certain printer functions, such as boldface and underlining. The enhanced printer driver uses the information in the printer action table (if present) to control the printing of these enhancements by sending the correct character sequences to the printer. A given character sequence may contain up to seven characters.

It's a good idea to have a copy of the printer manual nearby when executing BLDACT. The enhanced printer driver makes the following assumptions about a printer:

* Boldface and underlining can be turned on and off by character sequences.

* The printer can perform reverse linefeeds.

* The printer can set the amount of space a linefeed or reverse linefeed causes the platen to move.

Some printers will not be able to support the full range of enhancements. For example, on printers that cannot underline directly, a popular underlining method is to backspace and overstrike with an underbar. This methodology is not supported.

When BLDACT is executed, the following screen appears:

```
+-----------------------------------------------------------+
| BLDACT [1.0b] Build Printer Action Table                  |
| (c) Copyright 1983 Corvus Systems Inc.                    |
|                                                           |
| Character sequences to perform                            |
|                                                           |
|          Underline on.....                                |
|          Underline off.....                               |
|          Bold on.....                                     |
|          Bold off.....                                    |
|          Reverse line feed.....                           |
|          Back space.....                                  |
|                                                           |
| Superscript and subscript control sequences               |
|                                                           |
| 6 LPI :  Normal form advance distance.....                |
|          Sub and superscript form advance...              |
|                                                           |
| 8 LPI :  Normal form advance distance.....                |
```

```
¦          Sub and superscript form advance...                  ¦
¦                                                               ¦
¦ Proportional spacing control sequences                        ¦
¦                                                               ¦
¦ 10 CPI : Normal character spacing.....                        ¦
¦          Plus 1/120th of an inch spacing....                  ¦
¦          Plus 2/120th of an inch spacing....                  ¦
¦          Plus 3/120th of an inch spacing....                  ¦
¦                                                               ¦
¦ 12 CPI : Normal character spacing.....                        ¦
¦          Plus 1/120th of an inch spacing....                  ¦
¦          Plus 2/120th of an inch spacing....                  ¦
¦          Plus 3/120th of an inch spacing....                  ¦
+---------------------------------------------------------------+
```

```
    F1        F2        F3        F4        F5
+-------------------------------------------------+
¦  Prev  ¦  Next  ¦         ¦ DelChar¦ClrField¦
+-------------------------------------------------+

    F6        F7        F8        F9       F10
+-------------------------------------------------+
¦ReadFile¦         ¦WritFile¦         ¦  Exit  ¦
+-------------------------------------------------+
```

The cursor will be positioned at the top of the form, just by the Underline On position. The idea is to type in the characters needed to perform each function. The cursor can be moved up or down with either the arrow keys or the [Prev] and [Next] keys.

For example, if the printer is use takes Escape-E as a command to turn on boldface, moving the cursor next to the Bold On position and pressing the "Escape" and "E" keys would show:

```
¦          Bold on.............ESCAPE "E"                       ¦
```

The left and right arrow keys can be used to move the cursor to the various characters in a control sequence. [DelChar] and [ClrField] provide a simple editing capability that can be used to modify existing fields. Previously created tables may be edited by reading them into the utility with the F6 key.

After the action table is finished, save it back out to the /CCUTIL volume, preferably with an .ACT extension to distinguish it as an action table.

Before an action table can be used, it must be connected to the enhanced printer driver. The SETDTACOM function of SysUtils is used to perform this function. From the dispatcher, type:

```
[F5]    - Invoke the systems utilities
[F3]    - SetDtaCm
[F7]    - PtrFunc
[F5]    - LdAction
```

The system will prompt for the name of the action table file. After it's entered, pressing [F10] a few times will get out of everything, and the printer will be ready to use.

An alternate method of attaching an action table is to use the ACTION keyword in the CC.SETDCP utility. This can be put into a startup file:

```
!CC.SETDCP UNIT=PRINTER ACTION=<filename>
```

However, note that the SETDCP utility forces the printer to one or the other of the serial ports. If a printer is attached to a parallel card, this sequence would reassign the enhanced printer driver output to a serial port. In this case, execute the SETDCP utility before loading the parallel printer driver. The action table will remain active.

.DALT- The printer alternate character table utility

The enhanced printer driver supports an alternate character table. This allows the user to print special combinations of characters that are not part of the standard ASCII character set.

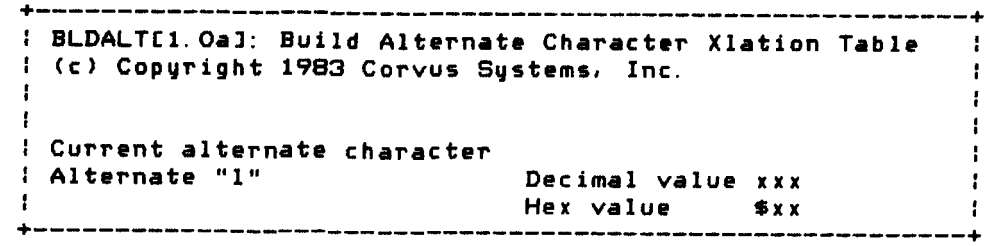
Alternate characters are those characters typed when the ALT key to the right of the space bar is held down. This sets the high order bit of the character's ASCII value to 1. If an alternate character translation table is in use, the enhanced printer driver will use the alternate character as an index into a table of strings, and the corresponding string will be printed in place of the alternate character.

For example, ALT-E could be used to specify the string "e", backspace, "'", to print an "e" with an accent mark over it. By using the character set editor to edit the appropriate character (ASCII of "E" + 128), it's possible for ALT-E to display correctly on the screen!

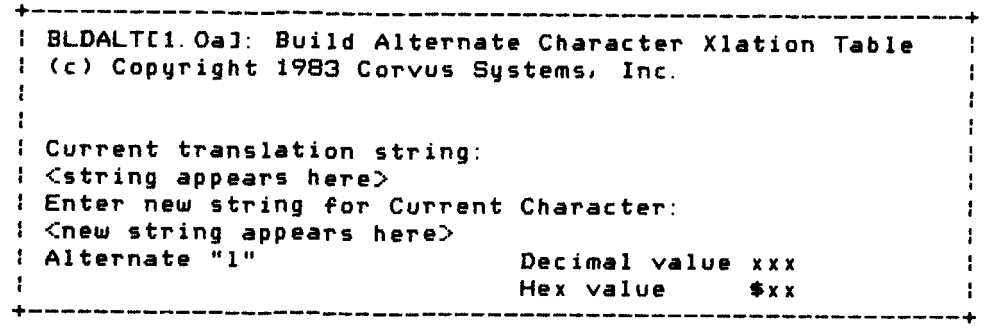
When invoked, the BLDALT utilitiy will display:

```
+-----------------------------------------------------------+
| BLDALT[1.0a]: Build Alternate Character Xlation Table     |
| (c) Copyright 1983 Corvus Systems, Inc.                   |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
+-----------------------------------------------------------+

+-----------------------------------------------------------+
| Enter alternate character or press function key:          |
+-----------------------------------------------------------+

    F1        F2        F3        F4        F5
+-------------------------------------------------+
|ReadFile| Select |ShowStrs|         |Dlet Str|
+-------------------------------------------------+

    F6        F7        F8        F9       F10
+-------------------------------------------------+
|WritFile|         |         |         |  Exit  |
+-------------------------------------------------+
```

To create a new alternate character table, hold that ALT key and press another key. Almost any key on the keyboard may be used. The screen will display:



After a character has been entered, pressing [Select] causes the system to prompt for the string to be sent in place of the alternate character. The string is entered by pressing the appropriate keys. The screen will display:



A maximum of nine characters may be in a string for any alternate character. After one alternate character has been defined, another is selected by pressing ALT and the desired character. [ShowStrs] will show all alternate characters and their associated strings; [DletStr] will delete the table entry for the current alternate character. [ReadFile] and [WritFile] load and save the alternate character tables, respectively. It's a good practice to use an .ALT extension on alternate character files.

BLDALT has no string editing facility; to change a string, reenter it in its entirety.

BLDCRT- The CRT Table Builder utility

The Concept display drivers DRV.DISPVT and DRV.DISPHZ are table driven- the control sequences they respond to are kept in the CRT table. Changing the CRT table changes the way the display will react to given control codes, and allows the system to emulate other types of terminals at the driver level. For example, a communications program that was used to talk to a computer configures for Lear Siegler terminal could just load the appropriately configured CRT action table. This methodology frees the applications program from having to handle the screen protocols.

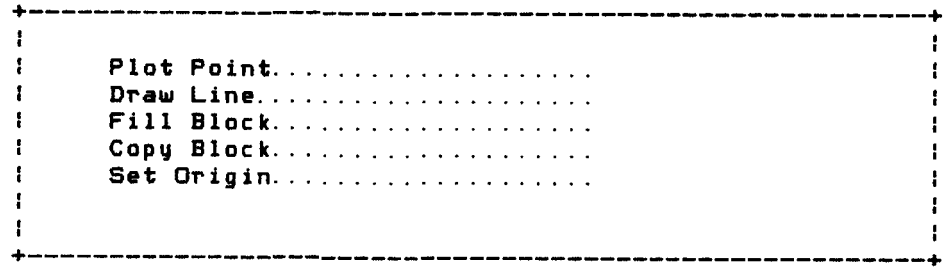
When it is executed, the BLDCRT utility displays:

```
+-----------------------------------------------------+
| BLDCRT [1.0]: Build a CRT table                     |
| (c) Copyright 1983 Corvus Systems, Inc.             |
|                                                     |
|   Common  - Edit common CRT functions               |
|   Cursor  - Edit cursor movement functions          |
|   Video   - Edit video functions                    |
|   Graphics- Edit graphics functions                 |
|   Misc    - Edit miscellaneous functions            |
|   Test    - Test current table                      |
|   ReadFile- Read file                               |
|   WritFile- Write current table to disk             |
|   Exit    - Exit                                    |
|                                                     |
+-----------------------------------------------------+

   F1       F2       F3       F4       F5
+-----------------------------------------+
|ReadFile|  Test  | Common |        |WritFile|
+-----------------------------------------+

   F6       F7       F8       F9      F10
+-----------------------------------------+
| Cursor | Video  |Graphics|  Misc  |  Exit  |
+-----------------------------------------+
```

When [Common], [Cursor], [Video], [Graphics], or [Misc] is pressed, the apprpriate functions will eb displayed on the screen, and the function keys will change to:

```
   F1       F2       F3       F4       F5
+-----------------------------------------+
|ReadFile|        | Clear  |        |WritFile|
+-----------------------------------------+

   F6       F7       F8       F9      F10
```

For example, pressing the [Graphics] key would display:



The cursor can be moved from one field to the next with the [Previous] and [Next] keys. Typing any other character causes that character to be entered at the cursor position. Each CRT function may be either one character (regular or control), or an Escape followed by a character. The character sequence is entered by typing it.

Character sequences cannot be edited, but must be replaced by pressing [Clear] and then entering the appropriate characters.

Pressing the [Exit] key at this point will return to the outer level of the program.

To attach a new function table to the display driver, a short Pascal program must be written:

```
Program LoadCRTTable;

Uses {$U /ccutil/cclib}
  CCdefn;

Const
  DispUnit = 36;

Var
  F     : File;
  FName : String[80];
  Blk   : Array[0..511] of Byte;

Begin
  Write('Enter name of CRT table file: ');
  ReadLn(FName);
```

```
    ReSet(F,FName);
    If IOResult <> 0 Then Begin
      WriteLn;
      WriteLn('No such file!');
      Exit(LoadCRTTable)
      End;
    Blks := BlockRead(F,Blk,1);
    If Blks <> 1 Then Begin
      WriteLn;
      WriteLn('Can''t read that file!');
      Exit(LoadCRTTable)
      End;
    UnitStatus(DispUnit,Blk,8)          { Attach! }
  End.
```

More information on CRT tables can be found in chapter 5.

Program segmentation

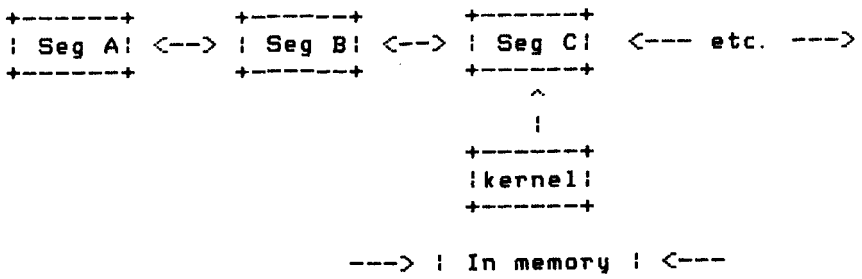
CCOS compilers support a segmentation directive that allows large programs to be broken up into segments. A segment is a collection of procedures that is either in memory, or not. When any procedure in a segment is called, the entire segment is swapped in from disk, if it is not already in memory. The maximum size of a segment is 32K words (64K bytes). Thus, programs larger than 64K bytes must be segmented.

A segment will remain in memory as long as any procedure in the segment is executing or being called. (A procedure can be called, but not executing, if it has in turn called another procedure.) As soon as no procedure in the segment is in use, it is swapped out. Once a segment has been swapped out, further references to any procedures in the segment force the segment to be called from disk.

Calling a segment from disk takes roughly 50ms per block, while calling a procedure in memory takes only 20us. Thus, careless segmentation of a program can result in excessive disk thrashing and consequent dramatic reductions in performance.

Ideally, a program's segments should divide the code into working sets. A working set is simply a collection of procedures that tend to get called in the same time frame. An example of procedure that belong in the same working set are the INSERTLINE and DELETELINE functions in an editor.

A simple segmentation scheme is:



In the above diagram, segments kernel and Seg C are resident in memory, while segments A, B, ... N are on disk. With this scheme, only two segments are ever in memory at one time. The program mainline is kernel, which always resides in memory, and calls the other segments. A Pascal program's mainline is always memory resident, since by

definition it is always either running or calling other procedures. If the kernel or any of the working segments exceed the 32K word segment size limit, then they must be segmented in turn.

Both Pascal and FORTRAN support segmentation. The segment scheme is controlled through compiler directives. Procedures and functions in a segment need not be contiguous in the program source. For example, in the following Pascal program:

```
{$S seg1}
Procedure Glotzy;
  <...>
Procedure Zilch;
  <...>

{$S seg2}
Procedure Hiccup;
  <...>
Function Qwerty: Integer
  <...>

{$S seg1}
Procedure Largesse;
```

the procedures Glotzy, Zilch, and Largesse would all be in segment seg1, while Hiccup and Qwerty would be in seg 2.

Segment locking

Normally, only 32K words of a program's code may be resident. In order to make use of the Concept's large memory, multiple segments may be locked in memory using the SegLock and SegUnLock procedures in CCLIB (the equivalent FORTRAN subroutines are SEGLCK and SEGUNL). There are some restrictions on the use of these procedures:

* Calls to SegLock and SegUnLock must be made from the segment being locked or unlocked.

* Segments may be locked only one time before being unlocked.

* Segments may be unlocked only one time before being locked again.

* Segments must be unlocked in the reverse order in which they were locked.

Following is a sample program demonstrating the use of SegLock and SegUnLock:

```
Program SegText;

Var
  J : Integer;
  Ch: Char;

Procedure SegLock; External;
Procedure SegUnLock; External;

{$S seg1}

Procedure LockIt;

Begin
  SegLock
End;

Procedure UnLockIt;

Begin
  SegUnLock
End;

Procedure Seg1A;

Begin
  WriteLn('Segment not locked')
End;

Procedure Seg1B;

Begin
  WriteLn('Segment locked')
End;

{-----------------------------------------------}

{$S}                    { Main segment }

Begin
  For J := 1 To 50 Do Seg1S;
  LockIt;
  For J := 1 To 50 Do Seg1B;
  UnLockIt;
  Read(Ch);             { Wait for user to hit key}
  WriteLn;
```

```
  For J := 1 To 50 Do Seg1A
End.
```

The Linker

The linker is a standard CCOS utility used to connect or link various object modules into a single, directly executable program. The OBJ files produced by the Pascal and FORTRAN compilers cannot be run, since they do not include the standard system I/O routines and whatever other special library units they might require. Additionally, they are not in the correct format, so even a completely standalone machine code routine would have to be put through a dummy link in order to work.

The linker operates by taking as input the pathnames of an arbitrary number of object files. The first file in the sequence is considered the "main" file. The linker attempts to complete unresolved references in the main file by finding the appropriate objects in the succeding files. The linker does type check the objects involved to make sure that the number and type of parameters in the main file's invocation match those of the actual object.

Below is an example of the dialogue created when the linker is invoked with no parameters:

```
+-----------------------------------------------------------+
| LINKER - MC68000 Object Code Linker 1.1  01-Dec-82        |
| (C) Copyright 1982 Silicon Valley Software, Inc.          |
|                                                           |
|                                                           |
| Listing file -                                            |
| Output file - Raskin                <-- Output file       |
| Input file [.OBJ] - Raskin          <-- Program file      |
| Input file [.OBJ] - /ccutil/cclib <-- System lib stuff    |
| Input file [.OBJ] - !paslib         <-- Pascal I/O stuff  |
| Input file [.OBJ] -                 <-- We're done        |
|                                                           |
| Linking segment "       "                                 |
| The output is executable.                                 |
+-----------------------------------------------------------+
```

The linker can optionally generate a listing file showing memory map information and various linker messages. In this example, the generation of a listing file was skipped by pressing [Return] without entering a file name. Note that all input files are assumed to have .OBJ extensions; thus, the output file name does not conflict with the first input file name, which is really RASKIN.OBJ.

The linker can also be invoked by entering all the necessary parameters on the command line. The parameter list consists of a list of pathnames. The first pathname is taken as the

first input file, and the output file name is generated by dropping the .OBJ from the input file name. The preceding example could be called in this fashion:

```
+-----------------------------------------------------------+
| Select function: linker raskin /ccutil/cclib !paslib      |
+-----------------------------------------------------------+
```

The final results would have been identical; only the messages output to the screen would be different.

Linker Options

The linker has several optional capabilities which can be entered on the command line when it is invoked. Linker options are single letter codes preceeded with a + sign to activate them or a - sign to deactivate them.

Option	Default	Function
P	-P	Display status information.
Q	+Q	-Q forces overlay format for the generated file. The standard is quick load format.
U	-U	Lists unreferenced entry points.
M	-M	Lists a memory map in the order in which the modules are linked.
A	+A	Lists memory map in alphabetical order.
A	-S	Shows all symbols starting with a % character. These symbols are generated by the compilers.

.inker error messages

There are three types of linker error messages. WARNINGS are correctable errors. If an input file name was misspelled and the linker couldn't find it, a

```
*** Warning - Can't open input file ***
```

message would be displayed. The user then has the opportunity to reenter the file name with the correct spelling.

ERRORS allow the user to proceed, although the resulting file will not be executable. An example would be an a file with an unresolved external reference.

FATAL ERRORS are those that the linker cannot recover gracefully from. In these cases the linker will either return to the dispatcher or kill the system.

Partial linking

Partially linked files are those with unresolved external references. Partially linked files can be used in subsequent link operations to form executable files.

Use of libraries

The intelligent and structured use of libraries can save a substantial amount of time during linking. A library is any collected group of routines. Examples of such libraries include the PASLIB library, which consists of Pascal I/O routines, and, more generally, CCLIB, which consists of various UNITs designed to make use of the Concept's features easier for the programmer.

When a library name is given to the Linker as part of the linking process, the Linker will automatically search the entire library in its attempt to satisfy unresolved external references in the host program. Thus, if you had a group of 10 machine code routines that you used often, you could use the Librian (see next section) to create a single library containing all these routines. Thus, a program that used six of these routines could link to all of them with a single file name, rather than forcing the user to type in six different file names.

For more information, see the section on the Library utility below.

Linking Pascal with FORTRAN with assembly...

Since all languages (currently) available under CCOS produce native code, it is theoretically and even practically possible to link segments produced by different languages into a single, executable program module.

The user who wishes to link modules from disparate languages must be cognizant of the different data structures supported

by the various languages. Most Pascal structures do not map directly to FORTRAN structures, and the interpretation of things such as sets (and other enumerated types) is the responsibility of the host program.

A intimate knowledge of the internal representation of Pascal and FORTRAN data structures is recommeded. The appropriate information may be found in the Pascal and FORTRAN manuals.

The Librarian

The Librarian is a utility whose purpose is to maintain groups of seperately compiled (or assembled) routines as libraries. Seperate compilation is a useful feature that allows routines that will be used often to be compiled (or assembled) once and then be available for linking.

The routines must be Pascal UNITs. Machine code or FORTRAN programs must therefore be called by a Pascal host. The structure of a UNIT is detailed in the Pascal reference manual.

Below is an example of a simple Librarian session. The Librarian utility is named LIBRARY.

```
LIBRARY - MC68000 Library Utility                01-Dec-82
(C) Copyright 1982 Silicon Valley Systems, Inc.

Listing file- LIBLST
Output file- /CCUTIL/NEWLIB.OBJ
Input file [.OBJ]- GRAF1
Input file [.OBJ]- GRAF2
Input file [.OBJ]-              < user presses RETURN >

Reading GRAF1...
Copying interface text of GRAF1
Reading GRAF2...
Copying interface text of GRAF2
```

Attempts to create libraries from routines that are not proper units will result in the rather cryptic message

Bad block 10 in <filename>

Once routines are collected in a library, the Linker will automatically search the library for functions and procedures external to a given host program. Linking can therefore be greatly simplified since a single library link can replace an arbitrary number of seperate links.

Running the compilers

The supplied CCOS compilers (Pascal and FORTRAN) read text files produced by EdWord and produce machine code modules which are linked with the appropriate external and run-time routines to create executable programs.

The compilers cannot read EdWord workpads directly; the program text must be moved outside the EdWord workspace with the [SaveFile] command from within EdWord. [SaveFile] automatically appends the necessary ".TEXT" extension to the file name.

Both compilers operate in the same fashion: the supplied text file is compiled to an intermediate code or I-code file. The compiler then invokes the code generate which produces 68000 machine code from the I-code file. There is only one code generator; it is used by both compilers on the system.

The I-code file is normally automatically deleted which the code file is generated. Under some circumstances, such as a power failure or system crash, the I-code file may be left open and in an indeterminate state. This situation is generally brought to the attention of

The compilers are invoked by typing:

```
Pascal <filename>

        or

Fortran <filename>
```

These invocations will expect a file called <filename>.TEXT and procedure a file called <filename>.OBJ. No listing file will be produced. These defaults may be overridden by specifying input, object, and listing files in the command line:

```
Pascal -i<inputfilename> -o<outputfilename>
       -l<listingfilename>
```

The same -i, -o, and -l specifiers may be used with the FORTRAN compiler.

```
(*****************************************************************************)
(*                                                                           *)
(*                          File:  OS:OS.GLOBALS.TEXT                        *)
(*                                                                           *)
(*              (C) Copyright 1981 Silicon Valley Software, Inc.             *)
(*                                                                           *)
(*                            All Rights Reserved.              30-Sep-81    *)
(*                                                                           *)
(*****************************************************************************)

{$R-}

unit globals;

interface

const BLOCKSIZE = 512;
      VIDLENGTH = 7;
      TIDLENGTH = 15;
      MAXDIR = 77;
      MAXDEV = 36;
      MINSLT = 1;
      MAXSLT = 5;
      MAXJTABLE = 40;
      MAXUTABLE = 10;
      MAXPROCESS = 10;
      MAXWINDOW  = 20;
      LOCODELOC  = $0108;
      HICODELOC  = $010C;
      SysComPLoc = $0180;
      SysKybdFlg = $0184;  {keyboard control flags}
      SysByteScn = $0186;  {display driver - bytes per scan line}
      CPtprvrs   = $070C;  {OMNINET Transporter version number}
      CPuserID   = $071C;  {user name field - decrypted}
      CPuserName = $0726;  {user name field - encrypted}

      FNORMAL = 0; FLOCK = 1; FPURGE = 2; FTRUNC = 3;

      LongStrMax    = 1030;

      Wsense   = 0;      {window functions}
      Wcreate  = 1;
      Wdelete  = 2;
      Wselect  = 3;
      Wclear   = 4;
      Wstatus  = 5;
      WwrBytes = 6;
      WrdBytes = 7;

      DTndev   = 0;      { no device                  }
      DTlocl   = 1;      { local disk                 }
```

```
        DTomni    = 2;      { OMNINET disk                }
        DTc8      = 3;      { Corvus 8" SSSD floppy disk  }
        DTc5      = 4;      { Corvus 5" SSSD floppy disk  }
        DTa5      = 5;      { Apple 5" floppy disk        }
        DTbank    = 6;      { Corvus BANK                 }
        DTf8      = 7;      { Corvus 8" DSDD floppy disk  }
        DTf5      = 8;      { Corvus 5" DSDD floppy disk  }
        DTf3      = 9;      { Corvus 3" DSDD floppy disk  }

{$P}
type string80 = string[80];
     dirrange = 0..MAXDIR;
     vid = string[VIDLENGTH];
     tid = string[TIDLENGTH];
     filekind = (UNTYPEDFILE, XDSKFILE, CODEFILE, TEXTFILE, INFOFILE,
                 DATAFILE, GRAFFILE, FOTOFILE, SECURDIR);
     daterec = packed record
                       year: 0..100; { 100 = temp file flag }
                       day: 0..31;
                       month: 0..12; { 0 = date not meaningful }
                     end;

     direntry = packed record
                        firstblock: integer;
                        nextblock: integer;
                        MarkBit: Boolean;
                        filler: 0..2047;
                        case fkind: filekind of
                          SECURDIR,
                          UNTYPEDFILE:
                            (dvid: vid;              { Disk volume name }
                             deovblock: integer;     { Last block of volume }
                             dnumfiles: integer;     { Number of files }
                             dloadtime: integer;     { Time of last access }
                             dlastboot: daterec;     { Most recent date setting }
                             MemFlipped: Boolean;    { TRUE if flipped in memory }
                             DskFlipped: Boolean);   { TRUE if flipped on disk }
                          XDSKFILE, CODEFILE, TEXTFILE, INFOFILE,
                          DATAFILE, GRAFFILE, FOTOFILE:
                            (dtid: tid;              { Title of file }
                             dlastbyte: 1..BLOCKSIZE; { Bytes in last block }
                             daccess: daterec);      { Last modification date }
                      end;

     directory = array[dirrange] of direntry;
     pdirectory = ^directory;

     devrange = 0..MAXDEV;

{$P}
     byte      = -128..127;
```

```
      bytes     = array [0..$7FFE] of byte;
      pBytes    = ^bytes;
      ppointer  = ^pBytes;
      words     = array [0..$7FFE] of integer;
      pWords    = ^words;
      lwords    = array [0..$7FFE] of longint;
      pLWords   = ^lwords;
      string2   = string[02];
      string16  = string[16];
      string19  = string[19];
      string32  = string[32];
      string64  = string[64];
      pstring64  = ^string64;
      str64rec  = record s: string64; end;
      pstr64rec = ^str64rec;
      stringtable = array[1..100] of pstr64rec;
      pstringtable = ^stringtable;
      str8      = string[8];
      pstr8     = ^str8;
      str16     = string[16];
      pstr16    = ^str16;
      str120    = string[120];
      pstr120   = ^str120;


      addrtable = array[0..MAXJTABLE] of pBytes;
      paddrtable = ^addrtable;

      uaddrtable = array[0..MAXUTABLE] of pBytes;
      puaddrtable = ^uaddrtable;

{$P}
      pDrvHdr    = ^DrvHdrRcd;
      DrvHdrRcd = record
{length offset}
{     1       0 }  bra:      byte;
{     1       1 }  ofs:      byte;
{     1       2 }  Blocked:  boolean;
{     1       3 }  comnds:   byte;
{     1       4 }  yr:       byte;
{     1       5 }  mo:       byte;
{     1       6 }  dy:       byte;
{     1       7 }  fill:     byte;
{     -       8 }  msg:      string64;
                   end;

      pdevtable = ^devtabrec;
      devtabrec =  record maxdevno: integer;
{length offset}           dt: array[devrange] of
{     2       0 }                 record comnds:  integer;
{     4       2 }                        driver:  pBytes;
```

```
{    1       6 }                        Blocked:  boolean;
{    1       7 }                        Mounted:  boolean;
{    8       8 }                        devname:  vid;
{    4      16 }                        devsize:  longint;
{    1      20 }                        devslt:   byte;     {device slot nmbr}
{    1      21 }                        devsrv:   byte;     {device server nmbr}
{    1      22 }                        devdrv:   byte;     {disk drive nmbr}
{    1      23 }                        devtyp:   byte;     {disk drive type}
{    1      24 }                        devspt:   byte;     {sectors per track}
{    1      25 }                        devtps:   byte;     {tracks per side}
{    1      26 }                        devro:    boolean;  {device read only}
{    1      27 }                        devflp:   boolean;  {volume directory flipped}
{    4      28 }                        devblk:   longint;  {disk base block}
{ total     32 }                  end;
                 end;

     SlotTypes = (NoDisk, LocalDisk, OmninetDisk,
                  FlpyC8Disk, FlpyC5Disk, FlpyA5Disk,
                  BankDisk,
                  FlpyF8Disk, FlpyF5Disk, FlpyF3Disk);

     pslttable = ^slttabrcd;
     slttabrcd = record
{length offset}
{    2       0 }  bootslt:  integer;
{    2       2 }  bootsrv:  integer;
{    2       4 }  actslt:   integer;
{    2       6 }  actsrv:   integer;
{    2       8 }  altslt:   integer;
{    2      10 }  altsrv:   integer;
{   20      12 }  st:       array [MINSLT..MAXSLT] of record
                                slt:  MINSLT..MAXSLT;
                                typ:  slottypes;
                                ndr:  INTEGER;
                                end;
{   20      32 }  drv:      array [MINSLT..MAXSLT] of pBytes;
                  end;

{$P}
     pCharSet  = ^CharSet;
     CharSet   =  record
{length offset}
{    4       0 }   tblloc:  pBytes;   {character set data pointer}
{    2       4 }   lpch:    integer;  {scanlines per character (assume wide)}
{    2       6 }   bpch:    integer;  {bits per character (vertical height)}
{    2       8 }   frstch:  integer;  {first character code - ascii}
{    2      10 }   lastch:  integer;  {last character code - ascii}
{    4      12 }   mask:    longint;  {mask used in positioning cells}
{    1      16 }   attr1:   byte;     {attributes}
                                      {  bit 0 = 1 - vertical orientation}
{    1      17 }   fill1:   byte;     {currently unused}
```

```
{ total    18 }   end;

     pWndStat   = ^WndStat;
     WndStat    =  record
{length offset}
{    2       0 }   homex:   integer; {relative to current character set}
{    2       2 }   homey:   integer; {relative to current character set}
{    2       4 }   width:   integer; {relative to current character set}
{    2       6 }   lngth:   integer; {relative to current character set}
{    1       8 }   active:  boolean; {active window flag}
{    1       9 }   fill1:   byte;    {currently unused}
{ total    10 }   end;

     pWndRcd    = ^WndRcd;
     WndRcd     =  record
{length offset}
{    4       0 }   charpt:  pCharSet;{character set record pointer}
{    4       4 }   homept:  pBytes;  {home (upper left) pointer}
{    4       8 }   curadr:  pBytes;  {current location pointer}
{    2      12 }   homeof:  integer; {bit offset of home location}
{    2      14 }   basex:   integer; {home x value, rel to root window}
{    2      16 }   basey:   integer; {home y value, rel to root window}
{    2      18 }   lngthx:  integer; {maximum x value, bits rel to window}
{    2      20 }   lngthy:  integer; {maximum y value, bits rel to wimdow}
{    2      22 }   cursx:   integer; {current x value, bits rel to window}
{    2      24 }   cursy:   integer; {current y value, bits rel to window}
{    2      26 }   bitofs:  integer; {bit offset of current address}
{    2      28 }   grorgx:  integer; {graphics - origin x, bits rel to home}
{    2      30 }   grorgy:  integer; {graphics - origin y, bits rel to home}
{    1      32 }   attr1:   byte;    {inverse, underscore, insert}
{    1      33 }   attr2:   byte;    {v/h, graphics/char, cursor on/off,
                                      cursor inv/underline}
{    1      34 }   state:   byte;    {used for decoding escape sequences}
{    1      35 }   rcdlen:  byte;    {window description record length}
{    1      36 }   attr3:   byte;    {enhanced character set attributes}
{    1      37 }   fill1:   byte;    {currently unused}
{    1      38 }   fill2:   byte;    {currently unused}
{    1      39 }   fill3:   byte;    {currently unused}
{    4      40 }   fill4:   longint; {currently unused}
{    4      44 }   wwsptr:  pBytes;  {window working storage pointer}
{ total    48 }   end;

     pWndTbl    = ^WndTbl;
     WndTbl     =  array [0..MAXWINDOW] of pWndRcd;

{$P}
     pprocrec = ^procrec;
     procrec = record d: array[0..7] of longint;
                      a: array[0..7] of longint;
                      no: integer; iores: integer;
                      proresult: integer; statusreg: integer;
```

```
                        pc: longint;
                        hicode: longint;
                        locode: longint;
                        hidata: longint;
                        lodata: longint;
                        heaptop: longint;
                        jumptab: longint;
{cc}                    loheap: longint;
{cc}                    wndptr: pWndRcd;
{cc}                    pgmid: string19;
                end;

     pproctable = ^proctable;
     proctable = array[0..MAXPROCESS] of procrec;

     memrec = record lodata: longint;
                     hidata: longint;
                     locode: longint;
                     hicode: longint;
                     btsw:   integer;
                     btdev:  integer;
                     btslt:  integer;
                     btsrv:  integer;
                     btdrv:  integer;
                     btblk:  longint;
                     end;

     pfib = ^fib;
     fib = record fwindow: pBytes;
                  FEOLN: Boolean;
                  FEOF: Boolean;
                  FText: Boolean;
                  fstate: (FTVALID,FIEMPTY,FIVALID,FTEMPTY);
                  frecsize: integer;
                  case FIsOpen: Boolean of
                    TRUE: (FIsBlocked: Boolean;
                           funit: integer;
                           fvid: vid;
                           frepeatcount,
                           fnextblock,
                           fmaxblock: integer;
                           FModified: Boolean;
                           fheader:  direntry;
                           case FSoftBuf: Boolean  of
                             TRUE: (fnextbyte,fmaxbyte: integer;
                                    FBufChanged: Boolean;
                                    fbuffer: array[0..511] of byte;
                                    fuparrow: integer));
          end;

     ptext = ^text;
```

```
{$P}
     psyscom = ^syscomrec;
     syscomrec = record
{length offset}
{    2      0 }  sioresult: integer;
{    2      2 }  processno: integer;
{    4      4 }  freeheap: pBytes;
{    4      8 }  jtable: paddrtable;
{    4     12 }  sysout: ptext;
{    4     16 }  sysin: ptext;
{    4     20 }  sysdevtab: pdevtable;
{    4     24 }  pdirname: pstring64;
{    4     28 }  utable: puaddrtable;
{    2     32 }  today: daterec;
{    4     34 }  codejtaddr: longint;
{    2     38 }  nextprono: integer;
{    2     40 }  numpros: integer;
{    4     42 }  protable: pproctable;
{    4     46 }  pbootname: pstring64;
{    4     50 }  memmap: ^memrec;
{    2     54 }  bootdev: integer;      {boot device number}

                 {CONCEPT additions}
{    4     56 }  tempWndRcd: pWndRcd;   {temporary window record pointer}
{    4     60 }  syssltab: psltable;    {slot table pointer}
{    4     64 }  nextWndRcd: pWndRcd;   {next window record pointer}
{    4     68 }  currWndRcd: pWndRcd;   {current window record pointer}
{    4     72 }  currKbdRcd: pBytes;    {current keyboard record pointer}
{    2     76 }  UserID: integer;       {Constellation user ID}
{    4     78 }  vrsnmbr: pstring64;    {current version nmbr string pointer}
{    4     82 }  vrsdate: pstring64;    {current version date string pointer}
{    4     86 }  WndowTbl: pWndTbl;     {window table pointer}
{    2     90 }  suspinh: integer;      {suspend inhibit count}
{    2     92 }  suspreq: integer;      {suspend requested if non-zero}
{    1     94 }  TitleVol: byte;        {title line offset for volume}
{    1     95 }  TitleTim: byte;        {title line offset for time}
{    4     96 }  ppvolname: pstring64;  {program volume name pointer}
{ total   100 }  end;

{$P}
    SndRcvStr    = RECORD
                     sln: INTEGER; {send length}
                     rln: INTEGER; {recv length}
                     CASE integer OF
                         1: (c:   PACKED ARRAY [1..LongStrMax] OF CHAR);
                         2: (b:          ARRAY [1..LongStrMax] OF byte);
                         3: (str: PACKED ARRAY [1..LongStrMax] OF CHAR);
                         4: (int:        ARRAY [1..LongStrMax] OF byte);
                     END;
```

```
    Host_types  = (User_station,
                   File_server,
                   Printer_server,
                   Name_server,
                   Modem_server,
                   DB_server,
                   ON_interconnect,
                   X25_gateway,
                   SNA_gateway);

implementation

end. {globals}
```

Attaching Drivers to the Enhanced Printer Driver

The enhanced printer driver can send its output to any driver on the system. The following example Pascal program attaches a new printer driver called DRV.APIO that runs a printer from an interface card in one of the Concept slots. Normally, the output of the enhanced printer driver is sent to DTACOM2.

```
Program SetAPIO;

Uses
  {$U /ccutil/cclib} ccDefn;

{ Load the enhanced printer driver DRV.EPRNT, then
  load the new printer driver DRV.APIO.
  Attach the two so that output from DRV.EPRNT goes to
  DRV.APIO and thence to printer. }

Const
  RemOut = 8;
  Printer= 6;
  PrtUnit= 'Printer';

Var
  I,
  Unbr    : Integer;
  P       : pString80;
  MP      : pBytes;
  Mounted: Boolean;

{-------------------------------------------------------------}

Function pOSdevNam(U: Integer): pString80; External;

Procedure DoAssign(S1,S2,S3: String64);

Var
  ST : Array[1..10] of pString64;
  I  : Integer;

Begin
  St[1] := @S1; St[2] := @S2; ST[3] := @S3;
  I := Call('!ASSIGN',input,output,st,3);
  If I <> 0 Then Begin
    WriteLn('Error ',i:1,' assigning driver ',S1);
    Exit(SetAPIO)
    End
End;
```

```
Begin                              { Mainline }
  Mounted := False;
  P := pOSdevNam(Printer);
  If P^ = PrtUnit Then Begin       { See if printer mounted }
    MP := Pointer(Ord64(p)-1);
    Mounted := (MP^[0] = 1)
    End;
  If Not Mounted Then              { Not mounted }
    DoAssign('!DRV.EPRNT',PrtUnit,'6');
  P := pOSdevNam(RemOut);
  P^ := '';
  DoAssign('DRV.APIO','APIO','8');     { Assign new driver }
  Unbr := RemOut;
  UnitStatus(Printer, Unbr, $82);      { Attach to DRV.EPRNT}
  I := IORESULT;
  If I <> 0 Then WriteLn('Error ',i:1,' attaching drivers')
End.
```

Example timer driver routines

The following Pascal program and attached machine code comprise an example of a user timer routine. Once this routine is installed, the timer driver will call it at 50ms intervals.

```
Program timertest;

 const timer    = 34; {timer unit #}
       Bell     = 0;  {bell routine #}
       Create   = 1;  {Create routine #}
       Delete   = 2;  {delete rtn #}
       Disable  = 3;  {disable rtn #}
       Enable   = 4;  {enable rtn #}
       ONESHOT  = 2;  {continuous/1shot mode flag - 1 SHOT}
       SKIP1ST  = 4;  {skip first call flag TRUE}
       ok       = FALSE;

 type byte      = -128..127;
      bytes     = array[0..9999] of byte;
      proutine  = ^bytes;
      tblID     = integer;
      PBlockCre8 = record
                     case INTEGER of
                       0  : (BLAH : record
                                      address : proutine;
                                      count   : integer;
                                      flags   : integer;
                                      tableID : tblID
                                    end);
                       1  : (tableID : tblID);
                       2  : (BellBlock : record
                                           freq : integer;
                                           speaker : byte;
                                           fill : byte;
                                           duration : integer
                                         end);
                   end;

  var ParameterBlockCre8 : PBlockCre8;
      RestParameterBlock : PBlockCre8;
      ForBell            : PBlockCre8;
      i,j,a,q            : integer;
      answer             : string[80];

procedure DoNada; external;       { This is the timer routine }

procedure Wait; external;

procedure Clear; external;
```

```
function Test : boolean; external;

procedure Hello;
  begin
   if q<>0 then begin
     writeln('Hello there!!!!  ',q);
     q := q+1
   end;
  end;

BEGIN
write('Want timer table test? (y/n)  '); readln(answer);
if(answer[1]='y')or(answer[1]='Y') then begin
  {call timer driver to create an entry}
  ParameterBlockCre8.BLAH.address := @DoNada;
  ParameterBlockCre8.BLAH.count := 1; {call every 50 milliseconds}
  ParameterBlockCre8.BLAH.flags := 0; {do a continuous entry}
  ParameterBlockCre8.BLAH.tableID := $00FF; {give value to make sure correct}
  writeln('address of ParameterBlockCre8 = ',ord4(@ParameterBlockCre8));
  with ParameterBlockCre8.BLAH do begin
    write('address = ',ord4(address),'  count = ',count);
    writeln('  flags = ',flags,'   tableID = ',tableID);
  end;
  UnitStatus(timer,ParameterBlockCre8,Create);
  writeln(' tableID = ',ParameterBlockCre8.BLAH.tableID);
  RestParameterBlock.tableID := ParameterBlockCre8.BLAH.tableID;

  Clear;
  for i := 1 to 5 do begin
    Wait;
    writeln('Flag set ',i,' times');
  end;

  UnitStatus(timer,RestParameterBlock,Disable);

  Clear;  {in case interrupt during Disable operation}
  for i:= 1 to MAXINT do
    for j := 1 to 10 do;
  writeln('Waited ',ord4(MAXINT)*10,' iterations');
  if Test then writeln('After Disable flag set')
          else writeln('After Disable flag clear');

  UnitStatus(timer,RestParameterBlock,Enable);

  for i := 1 to 5 do begin     {flag should be clear from before Enable}
    Wait;
    writeln('After Enable, Flag set ',i,' times');
  end;

  UnitStatus(timer,RestParameterBlock,Delete);
```

```
    Clear;  {in case interrupt during Delete operation}
    for i:= 1 to MAXINT do;
    for j := 1 to 10 do;
    writeln('Waited ',ord4(MAXINT)*10,' iterations');
    if Test then writeln('After Delete flag set')
            else writeln('After Delete flag clear');

    {do timer table test with Pascal program}
      q := 0;    {make sure Hello doesn't write until ready}
      ParameterBlockCre8.BLAH.address := @Hello;
      ParameterBlockCre8.BLAH.count := 10; {call every 500 milliseconds}
      ParameterBlockCre8.BLAH.flags := 0; {do a continuous entry}
      ParameterBlockCre8.BLAH.tableID := $00FF; {give value to make sure corr
      writeln('address of ParameterBlockCre8 = ',ord4(@ParameterBlockCre8));
       with ParameterBlockCre8.BLAH do begin
        write('address = ',ord4(address),'  count = ',count);
        writeln('  flags = ',flags,'   tableID = ',tableID);
      end;
      UnitStatus(timer,ParameterBlockCre8,Create);
      writeln(' tableID = ',ParameterBlockCre8.BLAH.tableID);
      RestParameterBlock.tableID := ParameterBlockCre8.BLAH.tableID;

      q := 1; {allow Hello to start}
      a := 2; {wait till q=a}
      for i:=1 to 10 do begin
       while (q<a) do;    {wait for change}
       a := a+1;
      end;
      q := 0;{turn off Hello}
      UnitStatus(timer,RestParameterBlock,Delete);

  end;{timer table test}

write('Want Bell test? (y/n)  '); readln(answer);
if(answer[1]='y')or(answer[1]='Y') then begin
 repeat
   {make a bell sound}
   with ForBell.BellBlock do begin
     speaker := $55;
     fill := 0;
     duration := 10; {1/2 second}
     writeln( 'enter 0 to get defaults' );
     write('Enter speaker (a single byte) :  '); readln(a);
     if a<>0 then speaker := a;
     write('Enter duration in 50 millisecond ticks :  '); readln(a);
     if a<>0 then duration := a;
     freq := 1000;  {defaults}
     write('Enter frequency :  '); readln(a);
     if a<>0 then freq := a;
     write('Enter number of times : ');readln(a);
```

```
    end;

    for i:=1 to a do
     with ForBell.BellBlock do begin
       write('freq = ',freq,'   spkr = ',speaker);
       writeln('dur. = ',duration);
       freq := freq + 1;
       UnitStatus(timer,ForBell,Bell);
     end;
   write('Done? (y/n)  '); readln(answer);
  until ((answer[1]='y')or(answer[1]='Y'));
end;
END.
```

DoNada

Following is the source for a user routine to be serviced
by the timer driver. It is installed by the Pascal program
in the previous section.

```
;
; interrupt service routine called by timer interrupt service routine
; AND routines to verify interrupt occurred.
;
                global  DONADA,Wait,Clear,Test
;
stop1           equ     0               ;flag saying interrupt routine called
;
DONADA          lea     flag,A0         ;show Wait interrupt occurred
                bset    #stop1,(A0)
                rts
;
Wait            lea     flag,A0         ;wait until flag set
wait10          btst    #stop1,(A0)
                beq.s   wait10
                bclr    #stop1,(A0)     ;always clear when done
                rts
;
Clear           lea     flag,A0         ;clear interrupt occurred flag
                bclr    #stop1,(A0)
                rts
;
; test if stop flag set or clear
; function Test : boolean;
;
Test
                movea.l (SP)+,A1        ;get return address
                lea     flag,A0
                btst    #stop1,(A0)
                beq.s   test10
                move.w  #$01,(SP)       ;boolean is true if flag is set
                bra.s   test20
test10          move.w  #$00,(SP)       ;boolean is false if flag clear
test20          jmp     (A1)
flag            data    0               ;flag byte
;
                end
```