Chuck Hall's

MERLIN OPERATING SYSTEM

Interface Guide

First Edition

4th October 1981

Silicon Valley Software Incorporated
10340 Phar Lap Drive
Cupertino
California 95014

Table of Contents

PREFACE

MERLIN is a "mini" operating system for computer systems based on the Motorola MC68000 microprocessor.

The MERLIN Operating System Documentation is arranged into two distinct books.

The User's Guide is a "concepts and facilities" manual which explains the core ideas of MERLIN - its command interpreter, file system, and the utility commands that provide a means to get started on MERLIN. The User's Guide also contains information about the software packages and utilities that run under MERLIN. There are descriptions of how to run the compilers, the linker and librarian, and a summary of ED - the line-oriented editor.

The Internals Guide is a MERLIN Internal Interface Guide for programmers wishing to write software to run under MERLIN - it covers topics such as file structures, memory layout, device drivers, and other information about MERLIN.

There are other manuals in addition to these two. The additional manuals are whole, self-contained manuals such as the Pascal and FORTRAN reference manuals. These are separate because (a) they are large and placing them in the User's Guide would make that manual impossibly large, and (b) because they are separately priced-products.

Chapter 1

Introduction

MERLIN is a basic executive program for 68000-based microcomputer systems. Its main purpose is to provide an operating environment in which users can develop and run software applications quickly and easily. MERLIN's main features include:

- Single-user system - the user has the full power and responsiveness of the MC68000 system available with no competition for resources with other users.
- Fixed and demountable volumes (devices).
- Two level file structure.
- UNIX-like command language with re-direction of input and output.
- Automatic startup command file for initialization.
- The shell or command interpreter is simply a system command - users can develop their own shells to suit their specific needs.
- Assignable device drivers - new device drivers can be incorporated without the need for system reconfiguration.

Users view MERLIN as composed of several distinct parts:

- the file system provides a way to store data in named collections called files and a way to create, examine, remove, copy, and otherwise manipulate such files.
- the command interpreter, known as "the shell", provides the basic means of telling MERLIN what things it should do.
- the programming languages provide the means to write new software applications. MERLIN supports Pascal, FORTRAN, an

Chapter 2

General Information

This Chapter supplies general information about data structures and the means by which software makes MERLIN system calls. Topics covered in this Chapter are:

- a description of the units that MERLIN supports.
- data representation.
- various data structures such as the system communication area.
- memory layout, and program environment.

2.1 Units

MERLIN, as stated previously, looks somewhat like the UCSD Pascal system. MERLIN knows about several units, that is, external devices to or from which data may be transferred.

Generally speaking, it is only neccessary to be concerned with units when using unit input-output - the software layer below that of file input-output. The unit numbers that MERLIN currently deals with are as follows:

Unit Number and Name Description

0 - /null is a "null" device. It acts as an infinite sink or "black hole" when it is written to; when is is read from, an end-of-file condition is returned.

1 - /console is the console, that is, the keyboard and screen, with echo.

Characters, or bytes, occupy 16 bits if they are not packed. Packed characters occupy a byte and are aligned on a byte boundary.

Words occupy two bytes, or 16 bits. Words are the Pascal **integer** data types. Words are always aligned on a two byte boundary. Words represent signed integers in the range -32768 .. +32767.

Long Words occupy four bytes, or 32 bits. Long words are always aligned on a two byte boundary. Long words are accessible in Pascal by the **longint** data type. Long words represent signed integers in the range -2,147,483,648 .. +2,147,483,647. Long words are also used to store memory addresses and pointers in Pascal.

2.2.2 Boolean Data Type

The Pascal implementation has a **Boolean** data type. A **Boolean** is always represented in a single byte quantity. A value of 0 (zero) represents **false**. A value of 1 (one) represents **true**. No other values are valid. When a **Boolean** value is not an element of a packed data structure, a full byte of storage is used to facilitate access.

2.2.3 The NIL Pointer — 4 bytes

As mentioned above, the Pascal implementation uses a long word or 32-bit quantity to represent a pointer. One of the important pointers is the nil pointer which points to no data element (for example, used to indicate the end of a list). In this implementation, **nil** is represented by the value zero (0).

2.2.4 The String Data Type

Pascal has a dynamic sized string data type similar to that of the UCSD Pascal system. A string is a sequence of bytes in memory, with the first byte in the string containing the length of the string (not including the first byte). This means that the maximum string length is 255 bytes. A string value must be aligned on a word boundary.

2.2.5 Packed Array of Character

2.3 The System Communication Area

MERLIN maintains a System Communication Area in RAM. The System Communication Area contains global information that is important to running programs. Two of the important items are the "IORESULT", which is the return code from input-output operations, and the start address of the system call jump vector.

The System Communication Area base address is contained in the long word found in absolute location $180. The System Communication Area layout is described here.

IORESULT is a word value which contains a result code after completion of any input-output process.

PROCESS NUMBER is a word value, which is the current process number. The initial shell is assigned process number 0. Each subsequent process receives an incremented process number.

FREE HEAP is a long word pointer to the start of the free memory available for storage allocation.

SYSTEM CALL VECTOR
is a long word pointer to the start of the system call vector. The system call vector is a table of jump addresses to the system routines. This is described in more detail later on.

SYSOUT is a long word pointer to the initial shell's standard output file. SYSIN and SYSOUT are used for court of last resort error messages when the Pascal system runs into trouble, for example, when it runs short of allocatable storage.

SYSIN is a long word pointer to the initial shell's standard input file.

SYSTEM DEVICE TABLE
is a long word pointer to the device table.

DIRECTORY NAME is a long word pointer to the currently "logged" directory name.

byte +0	IORESULT
+2	Process Number
+4	Pointer to next available free space on the heap
+8	Pointer to start of System Call Vector
+12	Pointer to System Output File
+16	Pointer to System Input File
+20	Pointer to System Device Table
+24	Pointer to Boot Device Directory Name
+28	Pointer to Start of User Command Table
+32	Today's Date (held as a Packed Record)
+34	Overlay Jump Table Address
+38	Next Process Number
+40	Number of Processes
+42	Pointer to the Process Table Array
+46	Pointer to the Name of the Boot Device
+50	Pointer to Memory Bounds Map
+54	Boot Device Number

Figure 2-1
System Communication Area Layout

2.4 The System Call Vector

All MERLIN system calls are, at this time, made by reference

2.4.1 Calling a System Routine

To call a system routine, the appropriate parameters must be pushed onto the stack. The last thing pushed onto the stack should be the return address (normally pushed via a JSR instruction). The address of a system routine is extracted from the system-call vector, and a JSR to that address is then executed.

The code fragment below illustrates a way to call a system routine. In this specific example, the routine FCLOSE is called to close a file.

```
PEA        FBUFF        ;  Push address of FIB.
CLR.W      -(SP)        ;  Close type := NORMAL.
MOVE.L     $180.W,A0    ;  A0 := System Communication Area address.
MOVE.L     8(A0),A0     ;  A0 := System Call Vector address.
MOVE.L     32(A0),A0    ;  A0 := Address of FCLOSE entry.
JSR        (A0)         ;  Call the FCLOSE routine.
... Return Address ...;    FCLOSE returns to here
```

2.5 File Information Block (FIB)

Access to files requires passing the address of a File Information Block, abbreviated to FIB. A FIB contains all information about a file, its type, buffering and so on.

Before a file can be opened, an FIB must be allocated. The total number of bytes to be allocated depends on whether using Block input-output is being used. If Block input-output is being used, the FIB is 64 bytes long. In this case, the user must also allocate a buffer for the block. If Block input-output is not being used, in other words the file is a text file or an ISO file of type, the FIB is 576 bytes long, plus the number of bytes in a record.

WINDOW is a long word pointer to the file 'window' - the area at the end of the FIB that holds the current record.

END OF LINE is a Boolean that is true if an end-of-line was encountered in the file, false otherwise.

occupies 26 bytes in the FIB.

SOFT BUFFER is a **Boolean** quantity that when **true**, indicates that the file buffer for this file is actually a part of this structure, instead of separately allocated as in the case of a blocked file. When SOFT BUFFER is **true**, the following items are part of the File Information Block.

NEXT BYTE is a word quantity that is the next byte position to be read or written in the buffer.

MAXIMUM BYTE is a word quantity that is the number of the last byte in the buffer. This is used when reading a file that has a partial last block or when writing any file.

BUFFER CHANGED is a **Boolean** quantity that when **true**, indicates that the file buffer in this FIB has been changed and therefore must be eventually written back to the disk.

BUFFER is a 512 byte array - the size of one logical disk block.

RECORD WINDOW is an array of bytes sufficiently large to hold one record from the file. If that record is an odd number of bytes in size, the buffer is increased to be an even number of bytes long.

The diagram on the next page is a graphic layout of a File Information Block.

2.6 Device Directory

A directory resides on a blocked device. The device directory contains information about the volume and the files that reside on that volume. A complete directory is an array of 73 directory entries, the first entry being the header record which describes the specific volume. The other 72 entries are for the files that reside on the device. The elements in a directory entry are described here:

FIRST BLOCK is a word quantity which is the number of the first avaliable block on this device. This entry is normally zero (0).

NEXT BLOCK is a word quantity which is the number of the next available block after this entry. For the volume header entry, this is normally 6.

FILE KIND is a four-bit quantity which is the kind of file that this entry describes. The next two Subsections describe the different layouts of a directory entry depending on the file kind field. The values of file kind that are of interest are:

0 a directory header entry.

2 a code file.

3 a text file.

5 a data file.

8 is also a directory header entry.

the file kind entry is followed by 12 bits of unused space to fill up the word.

2.6.1 Directory Entry for a Header Record

If the FILE KIND field in the directory entry indicates that this entry is a directory header record, the following fields are valid:

correspond to a file entry.

```
                 +--------------------------------+
  Byte --> +0    |          FIRST BLOCK           |
                 +--------------------------------+
           +2    |          NEXT BLOCK            |
                 +-----------+--------------------+
           +4    | FILE KIND |      UNUSED        |
                 +-----------+--------------------+
      +-----------------------------+-----------------------------+
   +6 |      DISK VOLUME NAME       |          FILE NAME          | +6
      +-----------------------------+                             |
  +14 |         LAST BLOCK          |                             |
      +-----------------------------+                             |
  +16 |      NUMBER OF FILES        |                             |
      +-----------------------------+                             |
  +18 |         LAST ACCESS         |                             |
      +-----------------------------+                             |
  +20 |          LAST BOOT          |                             |
      +--------------+--------------+-----------------------------+
  +22 | MEM FLIPPED  | DISK FLIPPED |          LAST BYTE          | +22
      +--------------+--------------+-----------------------------+
  +24 |           UNUSED            |         LAST ACCESS         | +24
      +-----------------------------+-----------------------------+
```

Figure 2-3
Layout of a Directory Entry

8 this unit can perform a UNITBUSY operation.

16 this unit can perform a UNITSTATUS operation.

ADDRESS OF DRIVER is a long word pointer to the driver code for this device.

BLOCKED a **Boolean** which when **true**, indicates that this is a blocked device.

MOUNTED a **Boolean** which when **true**, indicates that this device is mounted (a driver is assigned to it).

DEVICE NAME an eight-byte field which is the name of the device. The first byte is the length of the string; the remaining seven bytes are the actual name of the device.

DEVICE SIZE is a word quantity which is the number of 512-byte blocks on this device. For an unblocked device, it is set to the maximum integer, 32767.

The layout of each entry in the device table is as shown below.

```
                 +----------------------------+
Offset  +0       |   Valid Operation Bits     |
                 +----------------------------+
        +2       | Pointer to Driver Routine  |
                 +-------------+--------------+
        +6       |   BLOCKED   |   MOUNTED    |
                 +-------------+--------------+
        +8       |  Device Name occupies      |
                 |       eight bytes          |
                 +----------------------------+
       +16       |        Device Size         |
                 +----------------------------+
```

Figure 2-5
Individual Device Table Entries

13 File Not Open - Attempt to operate on a closed file.

14 Bad Format - Non-numeric data read in an Integer or Real read operation.

15 Ring Buffer Overflow.

16 Write Protect - attempt to write to a write protected device.

17 Seek Error - Seek on a file that is not a text file or a blocked file. Also seek to a negative record number.

64 Device Error of unknown origin.

2.10 Register Usage in MERLIN

Registers A4 .. A7 are reserved for system use as follows:

A4 holds the address of the overlay jump table.

A5 holds the address of the user global data.

A6 holds the base address of the local stack frame. A6 is undefined for a procedure at the outermost (main) level.

A7 holds the current stack top address.

All other registers are CLOBBERED when system calls are made.

2.11 Environment of A Running Program

The diagram below shows the run-time environment pointed to by register A5.

```
                 +--------------------------------+
       (A5)+20   |   ARGC (argument count)        |
                 +--------------------------------+
       (A5)+16   |  ARGV (point to Arguments)     |
                 +--------------------------------+
       (A5)+12   |  Pointer to Standard Output    |
                 +--------------------------------+
       (A5)+8    |  Pointer to Standard Input     |
                 +--------------------------------+
       (A5)+4    |         Return Address         |
                 +--------------------------------+
(A5)  -------->  |         Old Copy of A5         |
                 +--------------------------------+
```

Figure 2-7
Environment of a Running Program

Chapter 3

System Calls

This Chapter provides a blow-by-blow description of the system call interfaces. In all cases, parameters are described in the order in which they must be pushed onto the stack. The last thing pushed onto the stack, in all cases, is the return address. The discussions below cover the following topics:

- Unit input-output.

- File input-output.

- Memory Management.

3.1 Unit input-output

Unit input-output is at the lowest level of the system input-output facilities. Unit input-output references the physical devices in terms of physical blocks (on a disk). There are five system interfaces for unit input-output, namely UNITREAD, UNITWRITE, UNITBUSY, UNITCLEAR and UNITSTATUS. They are described in the subsections that follow.

3.1.1 UNITREAD and UNITWRITE - Direct Unit Data Transfer

UNITREAD and UNITWRITE are used to transfer information between a memory buffer and a specific unit. Parameters are:

<u>unit</u> <u>number</u> a word quantity representing the physical unit number involved in the transfer.

<u>buffer</u> <u>address</u> a long word pointer to the memory buffer.

<u>byte</u> <u>count</u> a word quantity representing the number of bytes

<u>control</u> a word quantity representing a control parameter whose meaning is agreed upon between UNITSTATUS and any of its callers.

<u>3.2 File input-output</u>

This Section describes those facilities that deal with files. In order to use the File input-output facilities, it is neccessary to allocate a File Information Block (FIB). See Chapter 2 for the details of an FIB. If Blocked input-output is being used, a buffer must also be allocated for the data transfer operations. The buffer must be big enough to hold the number of blocks to be transferred at any time.

3.2.1 FINIT - Initialize a File

FINIT sets up a File Information Block when the file is opened. The Open File function (FOPEN) usually calls upon FINIT to do this. User programs do not normally need to call FINIT. Parameters are:

<u>Pointer to FIB</u> a long word pointer to a File Information Block.

<u>bytes in a record</u>
a word quantity. There are special meanings attached to this parameter if it is zero or negative. If positive, it represents the number of bytes per record in the file. If zero or negative, it has the following meanings:

0 this file is an interactive file - it is talking to a device such as a terminal. An interactive file is to all intents and purposes the same as a text file. There are some minor differences in the way that end-of-line is handled.

-1 this file is a UCSD Pascal compatible file. It is normally declared as just **file;** (an untyped file), as opposed to a file of **some-type;**. With this file organization, the user must provide

Mode a word quantity indicating the disposition of the file after it is closed. The modes are:

0 normal - if the file is an old file - it existed prior to this program run, it is saved (retained) in the file system. If the file is a new file - created during this program run, it is deleted or purged from the file system.

1 lock - makes a file permanent in the file system, regardless of any conditions mentioned in case (0) above.

2 purge - purges or removes this file from the file system when the file is closed.

3.2.5 READCHAR - Read a Character from a File

READCHAR reads a single character from a file. READCHAR only applies to interactive (mode 0), or text (mode -2) files. Parameters are:

Pointer to FIB a long word pointer to a File Information Block.

READCHAR returns a single byte value on the top of the stack.

3.2.6 WRITECHAR - Write a Character to a File

WRITECHAR writes a character to a file. There is a field width specification which can cause space filling. WRITECHAR only applies to interactive (mode 0), or text (mode -2) files. Parameters are:

Pointer to FIB a long word pointer to a File Information Block.

Character to be written is a byte.

Size a word quantity representing a field width. If size is greater than one, the character is preceded with size-1 spaces.

3.2.7 SEEK - Position to a Specific Record in a File

3.3 Memory Management

This section describes those MERLIN system calls dealing with dynamic allocation and de-allocation of memory. Memory Allocation is done on a heap. The heap grows upward from the end of the user program. The user stack grows downward from the top of memory. When the two collide, there is mutual annihilation.

3.3.1 NEW - Allocate Storage

NEW allocates storage on the heap. Parameters are:

Pointer to Storage
a long word pointer which points to another long word pointer. The second pointer receives the start address of the allocated storage, in the event that there is enough storage to allocate. Note that NEW always returns a pointer that is aligned to a word boundary.

Byte Count
a word quantity representing the number of bytes to be allocated. Note that if an odd number of bytes are requested, NEW rounds up to an even (word) number and allocates that number of bytes.

3.3.2 DISPOSE - De-Allocate Storage

DISPOSE currently acts as a no-op. It does not actually dispose of de-allocate storage as in some Pascal implementations. DISPOSE does, however, return a NIL pointer to the caller. Parameters are:

Pointer to Storage
a long word pointer that itself points to another long word pointer. This second pointer is the address of the region of storage to be de-allocated.

Byte Count
a word quantity representing the number of bytes to be freed. It must be the same number as that given to the NEW call as described above.

Device is Valid Indicator
a long word pointer to a **Boolean** quantity which is set to **true** is the device named by the first parameter above is actually on the system. If this parameter is assigned the value false, none of the previous three parameters are defined.

The interpretation of the various parameters of GETDIR is as follows:

- If Device-is-Valid is **false**, the device named by the first parameter is not on-line. In this case, none of the other parameters are meaningful.
- If Device-is-Valid is **true**, The Device-Number parameter is assigned the number of the unit associated with that volume.
- The Device-Blocked parameter is set to **false** if the device is not a blocked device (such as the /printer). In this case, the Directory parameter is meaningless. If the Device-Blocked parameter is set to **true**, the device is a blocked device, in which case the Directory parameter contains the directory read in from that volume.

4.1.1 Unit Driver Command Parameter

The Command passed in register D4.W describes what operation is to be performed. The command values are summarized here and described in greater detail below. When a given driver gets control, the caller has already verified (from the unit table) that this command is valid for this particular unit driver. The values of the command are:

0 Install the driver - perform any required initialization.

1 Read from the unit.

2 Write to the unit.

3 Clear the unit - reset it to its initial state.

4 Test if unit is busy.

5 Return status of unit.

6 Unmount the unit.

Install — When MERLIN installs a unit, either at boot time or when a unit is explicitly assigned, it is called with the install parameter. The unit can perform any initialization code neccessary to set up cyclic buffers, place interrupt vectors and so on.

Read and Write — Are self-explanatory.

Clear — Initializes the device - clear pending interrupts and such.

Busy — Check if the unit is ready for data transfer.

Status — Return the status of the unit. This operation is device dependent.

Unmount — Unmount the unit. This is called when the unit is re-assigned a new driver or is de-assigned. At this time the unit driver should perform any clean up or restoring of interrupt vectors that might be neccessary.

The next piece of code is the entry for a unit driver, illustrating how the various sections of the driver are called depending on the specific command.

```
;
;          Entry point for the UART Driver.
;
UARTDRIV
        CLR.W    D7                ;  IORESULT := 0.
        MOVE.L   D1,A0             ;  A0 := Data buffer address.
        LEA      URTTABL,A1        ;  A1 := Base address of offset table.
        LSL.W    #1,D4             ;  D4 := Command*2 for word count.
        MOVE.W   0(A1,D4.W),D4     ;  D4 := Offset from URTTABL.
        JMP      0(A1,D4.W)        ;  Go to appropriate driver.
;
URTTABL DATA.W   URTINST-URTTABL      ;  Install driver.
        DATA.W   URTRD-URTTABL        ;  Read from UART.
        DATA.W   URTWR-URTTABL        ;  Write to UART.
        DATA.W   URTCLR-URTTABL       ;  Clear UART.
        DATA.W   URTBSY-URTTABL       ;  Test if Busy.
        DATA.W   URTST-URTTABL        ;  Return status.
        DATA.W   URTUNMT-URTTABL      ;  Unmount driver.
```

The next few code sections illustrate the entry points and give a broad view of the operations performed.

```
;
;     Constants to define the UART base addresses.
;
UARTA    EQU  $600000          ;  UART A data register.
UARTAC   EQU  $600002          ;  UART A command register.
;
;
URTINST                               ;  URTINST - Install the Driver.
        MOVE     #UARTAC,A0           ;  A0 := UART A control register.
        MOVE.B   #18,(A0)             ;  Select register 0.
        MOVE.B   #18,(A0)             ;  Reset the whole UART.
        MOVE.B   #2,(A0)              ;  Select register 2.
        ....  more code to
           .... initialize the UART
        RTS                           ;  Return to the caller.
;
;
URTUNMT                               ;  URTUNMT - Unmount the driver.
        RTS                           ;  Nothing to do in this driver.
```

Chapter 5

Interface Definitions in Pascal

This chapter shows the Pascal type definitions, and the procedure interfaces, to MERLIN. The information given here is the Pascal representation of the narrative information in the preceding Chapters.

5.1 Basic Constant and Type Definitions

```
Const
  BLOCKSIZE   = 512;    { number of bytes in a disk block            }
  VIDLENGTH   = 7;      { number of characters in a volume name      }
  TIDLENGTH   = 15;     { number of characters in a file name        }
  MAXDIR      = 72;     { max number of directory entries/volume     }
  MAXDEV      = 20;     { max number of devices on the system        }
  MAXJTABLE   = 22;     { number of entries in system call table     }
  MAXUTABLE   = 10;     { number of entries in user call table       }
  MAXPROCESS  = 10;     { max number of processes allowed            }
  SYSCOMPLOC  = $0180;  { System Communication Area Pointer          }
  LOCODELOC   = $0108;  { Lowest memory location pointer             }
  HICODELOC   = $010C;  { Highest memory location pointer            }

                        { File disposition codes                     }
  FNORMAL     = 0;
  FLOCK       = 1;
  FPURGE      = 2;
  FTRUNC      = 3;

Type
  string80 = string[80];
  dirrange = 0 .. MAXDIR;
  vid = string[VIDLENGTH]; 8 bytes
  tid = string[TIDLENGTH];
```

5.1.1 Layout of the Date Record

```
Type
  daterec = packed record
              year : 0 .. 100;  {  100 => temporary file  }
              day : 0 .. 31;
              month : 0 .. 12;  [  0 => date not meaningful  }
            end;
```

5.1.2 Layout of a Directory Entry

```
Type                                                 normally zero first avail(
  direntry =                                         usually 6, 6 dir blocks
        packed record
          firstblock : integer; —
          nextblock : integer; —
          status : boolean; — ??
          filler : 0 .. 2047; —                              struct string
          case fkind : filekind of                                    eof condition
            SECURDIR, UNTYPEDFILE:
              (dvid : vid;            { (disk volume name)           }
               deovblock: integer;    { last block of volume         }
               dnumfiles: integer;    { number of files — on volume  }
               dloadtime: integer;    { time of last access — wrong? } P.
               dlastboot: daterec);   { most recent date setting     }
               MemFlipped: Boolean;   { TRUE if flipped in memory    }
               DskFlipped: Boolean;   { TRUE if flipped on disk      } internal
             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; ?

  byte = -128 .. 127; 8 bits
```

5.1.3 File Interface Block Definition

```
type
 pfib = ^fib;
 fib = record fwindow: pbytes;     byte pointe
         FEOLN: Boolean;
         FEOF: Boolean;
         FTEXT: Boolean;                                        ?
         fstate: (FTVALID, FIEMPTY, FIVALID, FTEMPTY);    text only
         frecsize: integer;
         case  FIsOpen:  Boolean of               2 byte
          true: (FIsBlocked: Boolean;
                 funit: integer;
                 fvid: vid;
                 frepeatcount,  not used
                 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));          X ?
      end;
                                        : array [0..?] of byte
                                     or   string[ ]
                                          or ?
```

```
Type
  pprocrec = ^procrec;

  procrec = record  d: array[0 .. 7] of longint; } registers
                    a: array[0 .. 7] of longint;
                    no: integer;
            end;

  pproctable = ^proctable;

  proctable = array[0 .. MAXPROCESS] of procrec;
```

5.2.2 File Input Output

```
Procedure  FINIT(f: pfib;  recbytes: integer);

procedure  FGET(f: pfib);

procedure  FPUT(f: pfib);

procedure  FOPEN(fpathname: pstring64;
                 f: pfib:
                 NewFlag: Boolean);

procedure  FCLOSE(f: pfib;  fmode: integer);

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, fblock: integer;
                  ReadFlag: Boolean): integer;
```

CORVUS CONCEPT

Linker Librarian Reference Manual

LINKER and LIBRARY UTILITY

Reference Manual

First Edition

22nd December 1981

Silicon Valley Software Incorporated
10340 Phar Lap Drive
Cupertino
California 95014

This Linker and Library Utility Reference Manual was produced by:

Jeffrey Barth, R. Steven Glanville and Henry McGilton.

Silicon Valley Software Incorporated
Publication Number 810601-01

Copyright (C) 1981 by Silicon Valley Software Incorporated

All rights reserved. No part of this document may be reproduced, translated, transcribed or transmitted in any form or by any means manual, electronic, electro-magnetic, chemical or optical without explicit written permission from Silicon Valley Software Incorporated.

Table of Contents

Chapter 1

Introduction

The Linker and Library utilities are a pair of complementary programs which aid in the process of generating executable programs under the MERLIN operating system.

The Linker links or binds relocatable object-code modules, and optional modules from libraries, to form a program which is executable.

The Library utility builds a library from relocatable object-code modules. Such a library can contain frequently used procedures (such as the mathematical functions of FORTRAN) which can be used in subsequent link processes.

1.1 Building an Executable Program

To get from the source text of a program to an executable object code file, the user must proceed as follows:

1. The source file is compiled or assembled. The result of compiling or assembling is a self-relocatable object-code file, along with listings and error diagnostics. This process continues until a "clean" compilation or assembly is obtained.

2. The relocatable object-code is linked, possibly including run-time support libraries, to generate executable code into a disk file.

3. The program can then be run (executed) on the machine simply by typing its filename.

The following chapters in this manual describe the Linker and Librarian object-code management system.

1.2 Overview and Layout of this Manual

Chapter 2 covers the Linker, its use, options and messages.

Chapter 3 describes the Library management utility and how to use it to build a library of relocatable object-code modules.

Chapter 4 is a detailed description of how object-code files are constructed, together with details of the various types of blocks that go to make an object-code file.

Chapter 2

Linker

The Linker is a utility which accepts files of relocatable object-code generated by the various compilers and assemblers, plus library files generated by the Library utility, and links or binds those into a form suitable for execution.

The Linker can also perform a partial link, where a collection of relocatable object-modules is bound into one file that can be used in future linking operations. This is described later on in this section.

As well as binding together relocatable modules from various language processors, the Linker can search libraries of commonly used functions, (such as the PASCAL run time environment), and link those modules that are referenced into the final loadable output file.

In order to link relocatable modules into an executable object-code file, the Linker needs the following pieces of information:

- The optional name of the listing file where the Linker messages and memory map information is to be listed. If no listing file name is given, no memory map information is generated.

- The name of the object-code file in which to write the final linked output.

- The name(s) of the file(s) from which the relocatable object-code is read.

- A list of one or more libraries which are to be used to satisfy external references within the object-code file.

A typical Linker run is shown below. Linker responses are in **bold face text**, and user input is underlined.

Example of Linker Usage

```
% linker
LINKER - MC68000 Object Code Linker
20-Jul-81
(C) 1981 Silicon Valley Software, Inc.

Listing File - /console
Output file[.OBJ] - myproglinked
Input file[.OBJ] - myprog
Input file[.OBJ] - paslib
Input file[.OBJ] -
     ..... Lots of Linker Messages .....
%
```

The Linker keeps prompting for more "Input files" until an empty line (carriage return) is entered. This enables the entry of a whole list of libraries as places from which to satisfy external references. The last one entered is usually the name of a run-time library (PASLIB in this example). A ".obj" suffix is added to all input filenames if it is omitted from the filename when entered.

If the Linker cannot find a specific input file, it displays a message to the effect:

```
*** Warning - Can't open input file ***
```

and repeats the prompt for an input file. The incorrect filename is simply ignored and the link can be completed with no adverse consequences.

2.1 Linker Options

Linker options are supplied on the command line when the Linker is called up. Linker options are introduced by a "+" sign, a "-" sign, followed by a letter, or a "?". The options are as follows:

? Display status information.

q The -q option disallows quick-load format for the executable object-code file, and forces overlay format. The +q option (the default) allows quick-load format.

u The +u option lists unreferenced entry points. The default is -u.

m The +m option prints the memory map in the order in which modules are linked. The default is -m.

a The +a option prints the memory map in alphabetical order. The default is +a.

s The +s option prints symbols that start with the "%" sign. Such symbols are used for compiler generated symbols. The default is -s or do not print "%" symbols.

2.2 Linker Error Messages

The Linker can display various error messages in the course of its operation. The error messages are self-explanatory. There are three grades of error messages, with different outcomes:

Warnings are correctable errors. The error can be corrected and the link proceeds. For example, misspelling a filename will result in a message to the effect that the file cannot be opened, at which point the filename can be retyped.

Errors are correctable in that the user can proceed with the link process, but the generated object-code file is not created properly.

Fatal errors are those from which the Linker cannot correct or recover. In those cases the linker returns to the shell.

2.3 Partial Linking

As mentioned above, the Linker can perform a partial link, where the final output is not neccessarily executable, but a collection of separate relocatable object-code files can be combined into one file. The resultant file can then be used as an input file in subsequent link operations. The output of a u The +u option lists unreferenced entry points. The default is **-u**.

m The **+m** option prints the memory map in the order in which modules are linked. The default is **-m**.

a The +a option prints the memory map in alphabetical order. The default is **+a**.

s The +s option prints symbols that start with the "**%**" sign. Such symbols are used for compiler generated symbols. The default is -s or do not print "**%**" symbols.

2.2 Linker Error Messages

The Linker can display various error messages in the course of its operation. The error messages are self-explanatory. There are three grades of error messages, with different outcomes:

Warnings are correctable errors. The error can be corrected and the link proceeds. For example, misspelling a filename will result in a message to the effect that the file cannot be opened, at which point the filename can be retyped.

Errors are correctable in that the user can proceed with the link process, but the generated object-code file is not created properly.

Fatal errors are those from which the Linker cannot correct or recover. In those cases the linker returns to the shell.

2.3 Partial Linking

As mentioned above, the Linker can perform a partial link, where the final output is not neccessarily executable, but a collection of separate relocatable object-code files can be combined into one file. The resultant file can then be used as an input file in subsequent link operations. The output of a partial link can have unsatisfied external references.

If, for any reason, the linked object file has not had all its external references satisfied, the linker displays a message to the effect:

The output is not executable

This message appears when external references are not satisfied. It may mean that a program was missing some subroutines from a library (maybe the user forgot to include the library in the link process), or it also can appear when doing a partial link, in which case the message is to be ignored, since the full link will be done at a later date.

Chapter 3

Library Utility

The Librarian binds compiled or assembled relocatable object-code modules into a collection called a library. The purpose of a library is to provide a repository for commonly used object modules that have to be present when linking (see the Linker description), such that the common modules end up bound together into the final executable code module.

The library utility typically wants the following pieces of information form the user:

- The name of the file which is to receive the listing (results and log) of the library process.
- The name of the file which is to contain the generated library when the library generation process is complete.
- The name(s) of file(s) (with the .obj) suffix, which contain the constituent parts of the library to be generated.

A typical Librarian session appears below. Note that Librarian responses are in bold face text and user inputs are underlined.

```
% library
LIBRARY - MC68000 Library Utility
20-Jul-81
(C) 1981 Silicon Valley Software, Inc.

Listing file - /console
Output File[.OBJ] - bodleian
Input file[.OBJ] - bookshelf
Input file[.OBJ] - stacks
Input file[.OBJ] -
 .....  Lots of interesting Librarian messages .....
%
```

If the Librarian cannot find the specified input file it issues a message to the effect:

```
     The file 'whatever.obj' can't be opened
```

Chapter 4

Object File Formats

This chapter describes the layout of the object-code files that the Linker and Librarian can process. The various code blocks are described in sufficient detail that a compiler writer can generate object-code that is acceptable to the Linker and Librarian.

4.1 Notation Used to Describe Object File Formats

The symbol "::=" is read as "defined to be". Where a whole list of objects appear to the right of a "pile" of "::=" signs, it implies a choice of any of the objects.

Objects enclosed in "angle brackets", "<" and ">" are syntactic objects which are defined in terms of other objects.

An object followed by an asterisk sign, "*", can be repeated "zero to many times" (the list of objects can be empty).

An object followed by a plus sign, "+", can be repeated "one to many times" (there must be at least one of that object).

4.2 Linker File Layout

This section is a description of the Linker File at the "top level".

```
<Link File>     ::= <Module File>
                ::=   <Library File>
```

```
               ::=    <Unit File>
               ::=      <Execute File>

<Module File>  ::= <Module>* EOF mark

<Library File> ::= <Library Module Block>+ <Library Entry Block>+
                   <Module>+  <Text Block>*  EOF Mark

<Unit File>    ::= <Unit Block> <Module>+ <Text Block> EOF Mark

<Execute File> ::= <Executable Block> <Module>*
               ::=   <Quick Load Block>

<Module>       ::= <Module Name Block> <Other Block>+ <End Block>

<Other Block>  ::=  Entry Block
               ::=  External Block
               ::=  Start Block
               ::=  Code Block
               ::=  Relocation Block
               ::=  Common Relocation Block
               ::=  Common Definition Block
               ::=  Short External Block
               ::=  Data Initialization Block
               ::=  FORTRAN data area definition block
               ::=  FORTRAN data area Initialization Block
               ::=  FORTRAN Data Area Reference Block
               ::=  FORTRAN Executable Data Area Initialization Block
               ::=  FORTRAN Executable Data Area Reference Block
```

4.3 Byte Level Description of Linker Blocks

All Linker and Librarian object-code blocks start with a single "identifier byte". This block identifier takes values from 80 (base 16) upwards. The choice of values greater than 80 (base 16) is an attempt to minimise the probability that a regular ASCII text file is mistaken for the start of an object-code block.

4.3.1 80 - Module Name Block

```
                +--------+--------+--------+--------+
byte -->      0 |   80   |       size (3 bytes)     |
                +--------+--------+--------+--------+
              4 |            module name            |
                |             (8 bytes)             |
                +--------+--------+--------+--------+
             12 |           segment name            |
                |             (8 bytes)             |
                +--------+--------+--------+--------+
             20 |          csize (4 bytes)          |
                +--------+--------+--------+--------+
             24 | comments (24 .. size-1 bytes) ... |
                +--------+--------+--------+--------+
```

80	Hexadecimal 80 indicates a Module Name Block.
size	Number of bytes in this block.
module name	Blank padded ASCII name of module.
segment name	ASCII name of segment in which this module will reside.
csize	Number of bytes in the code block for this module.
comments	Arbitrary information - ignored by the Linker.

Overlay segment?

filled in how?

4.3.2 81 - End Block

```
                  +--------+--------+--------+--------+
byte -->        0 |   81   |     size (3 bytes)       |
                  +--------+--------+--------+--------+
                4 |          csize (4 bytes)          |
                  +--------+--------+--------+--------+
```

81 Hexadecimal 81 indicates this is an End Block.

size Number of bytes in this block - it is always 000008.

csize Number of bytes in the code block for this module.

4.3.3 82 - Entry Point Block

```
                 +--------+--------+--------+--------+
byte -->       0 |   82   |      size (3 bytes)      |
                 +--------+--------+--------+--------+
               4 |            link name              |
               8 |            (8 bytes)              |
                 +--------+--------+--------+--------+
              12 |            user name              |
                 |            (8 bytes)              |
                 +--------+--------+--------+--------+
              20 |          loc (4 bytes)            |
                 +--------+--------+--------+--------+
              24 | comments (24 .. size-1 bytes) ... |
                 +--------+--------+--------+--------+
```

82 — Hexadecimal 82 indicates this is an Entry Point Block.

size — Number of bytes in this block.

link name — Blank padded ASCII Linker name of entry point.

~~[illegible]~~?

user name — Blank Padded ASCII user name of entry point.

filled in how?

does this imply links within another name beyond what is given by language processor?

loc — Location of entry point relative to this module.

comments — Arbitrary information - ignored by the Linker.

4.3.4 83 - External Reference Block

```
                   +--------+--------+--------+--------+
byte -->        0  |   83   |     size (3 bytes)       |
                   +--------+--------+--------+--------+
                4  |             link name             |
                8  |             (8 bytes)             |
                   +--------+--------+--------+--------+
               12  |             user name             |
                   |             (8 bytes)             |
                   +--------+--------+--------+--------+
               20  |          ref 1 (4 bytes)          |
                   +--------+--------+--------+--------+
               24  |          ref 2 (4 bytes)          |
                   +--------+--------+--------+--------+
                   |               . . .               |
                   +--------+--------+--------+--------+
                   | each reference consumes 4 bytes   |
                   +--------+--------+--------+--------+
                   |               . . .               |
                   +--------+--------+--------+--------+
           16+4*n  |          ref n (4 bytes)          |
                   +--------+--------+--------+--------+
```

83 — Hexadecimal 83 indicates this is an External Reference Block.

size — Number of bytes in this block.

link name — Blank padded ASCII Linker name of external reference.

user name — Blank padded ASCII user name of external reference.

See previous Comment

ref 1 — Location of first reference relative to this module.

ref 2 — Location of second reference relative to this module.

. . . — Other references.

ref n — Location of last reference relative to this module.

4.3.5 84 - Starting Address Block

```
                +--------+--------+--------+--------+
byte -->      0 |   84   |     size (3 bytes)       |
                +--------+--------+--------+--------+
              4 |           start (4 bytes)         |
                +--------+--------+--------+--------+
              8 |           gsize (4 bytes)         |
                +--------+--------+--------+--------+
             12 | comments (12 .. size-1 bytes) ... |
                +--------+--------+--------+--------+
```

84	Hexadecimal 84 indicates this is a Starting Address Block.
size	Number of bytes in this block.
start	Starting address relative to this module.
gsize	Number of bytes in the global data area.
comments	Arbitrary information - ignored by the Linker.

4.3.6 85 - Code Block

```
                +--------+--------+--------+--------+
byte -->      0 |   85   |     size (3 bytes)       |
                +--------+--------+--------+--------+
              4 |           addr (4 bytes)          |
                +--------+--------+--------+--------+
              8 | object-code (8..size-1 bytes) ... |
                +--------+--------+--------+--------+
```

85	Hexadecimal 85 indicates this is a Code Block.
size	Number of bytes in this block.
addr	Module-relative address of first code byte.
object-code	The object-code - always an even number of bytes.

4.3.7 86 - 32-Bit Relocation Block

```
                     +--------+--------+--------+--------+
byte -->           0 |   86   |     size (3 bytes)       |
                     +--------+--------+--------+--------+
                   4 |          addr 1 (4 bytes)         |
                     +--------+--------+--------+--------+
                  12 |          addr 2 (4 bytes)         |
                     +--------+--------+--------+--------+
                     |                . . .              |
                     +--------+--------+--------+--------+
                  16 |    each addr consumes 4 bytes     |
                     +--------+--------+--------+--------+
                     |                . . .              |
                     +--------+--------+--------+--------+
             12+4*n  |          addr n (4 bytes)         |
                     +--------+--------+--------+--------+
```

86 Hexadecimal 86 indicates this is a 32-bit Relocation Block.

size Number of bytes in this block.

addr 1 Location of first address to relocate.

addr 2 Location of second address to relocate.

. . . Locations of other addresses to relocate.

addr n Location of last address to relocate.

4.3.8 87 - Common Block Reference

```
                  +--------+--------+--------+--------+
byte -->       0  |   87   |     size (3 bytes)       |
                  +--------+--------+--------+--------+
               4  |           common name             |
                  |            (8 bytes)              |
                  +--------+--------+--------+--------+
              12  |          ref 1 (4 bytes)          |
                  +--------+--------+--------+--------+
              16  |          ref 2 (4 bytes)          |
                  +--------+--------+--------+--------+
              20  |               . . .               |
                  +--------+--------+--------+--------+
                  | each reference consumes 4 bytes   |
                  +--------+--------+--------+--------+
                  |               . . .               |
                  +--------+--------+--------+--------+
           8+4*n  |          ref n (4 bytes)          |
                  +--------+--------+--------+--------+
```

87 — Hexadecimal 87 indicates this is a Common Block Reference.

size — Number of bytes in this block.

common name — Blank padded ASCII common block name.

ref 1 — Location of first reference relative to this module.

ref 2 — Location of second reference relative to this module.

. . . — Other references relative to this module.

ref n — Location of last reference relative to this module.

4.3.9 88 - Common Block Definition

```
                 +--------+--------+--------+--------+
byte -->       0 |   88   |     size (3 bytes)       |
                 +--------+--------+--------+--------+
               4 |           common name             |
                 |            (8 bytes)              |
                 +--------+--------+--------+--------+
              12 |          dsize (4 bytes)          |
                 +--------+--------+--------+--------+
              16 | comments (16 .. size-1 bytes) ... |
                 +--------+--------+--------+--------+
```

Why not combine this with previous?

88 Hexadecimal 88 indicates this is a Common Block Definition.

size Number of bytes in this block.

common name Blank padded ASCII common data area name.

dsize Number of bytes in this common data area.

comments Arbitrary information - ignored by the Linker.

4.3.10 89 - Short External Reference Block

```
                  +--------+--------+--------+--------+
byte -->      0   |   89   |      size (3 bytes)      |
                  +--------+--------+--------+--------+
              4   |            link name              |
                  |            (8 bytes)              |
                  +--------+--------+--------+--------+
             12   |            user name              |
                  |            (8 bytes)              |
                  +--------+--------+--------+--------+
             20   | ref 1 (2 bytes) | ref 2 (2 bytes) |
                  +--------+--------+--------+--------+
        18+2*n    |      . . .      | ref n (2 bytes) |
                  +--------+--------+--------+--------+
```

89 — Hexadecimal 89 indicates this is a Short External Reference Block.

size — Number of bytes in this block.

link name — Blank padded ASCII Linker name of external reference.

user name — Blank padded ASCII user name of external reference.

ref 1 — Location of first reference relative to this module.

ref 2 — Location of second reference relative to this module.

. . . — Locations of other references relative to this module.

ref n — Location of last reference relative to this module.

The Linker does not yet support the short external reference block. It is intended to provide for one-word offsets that are either filled in with call-relative, short-absolute calls, or possibly calls indexed by an A-register, probably A4. The Linker will support this type of block in the future, and compilers will have an option to control the kind of generated call.

4.3.11 8A - FORTRAN Data Area Definition Block

```
                 +--------+--------+--------+--------+
byte -->       0 |   8A   |      size (3 bytes)      |
                 +--------+--------+--------+--------+
               4 |          data area name           |
                 |             (8 bytes)             |
                 +--------+--------+--------+--------+
              12 |          dsize (4 bytes)          |
                 +--------+--------+--------+--------+
```

8A — Hexadecimal 8A indicates this is a FORTRAN Data Area Definition Block.

size — Number of bytes in this block.

data area name — Blank padded ASCII name of FORTRAN fixed data area.

dsize — Size of this data area.

4.3.12 8B - FORTRAN Data Area Initialization Block

```
                 +--------+--------+--------+--------+
byte -->      0  |   8B   |     size (3 bytes)       |
                 +--------+--------+--------+--------+
              4  |          data area name           |
                 |            (8 bytes)              |
                 +--------+--------+--------+--------+
             12  |          daddr (4 bytes)          |
                 +--------+--------+--------+--------+
             16  | data occupies bytes 16 .. size-1  |
                 | in the rest of the block |  00 *  |
                 +--------+--------+--------+--------+
```

Again, could be combined with previous

8B Hexadecimal 8B indicates this is a FORTRAN Data Area Initialization Block.

size Number of bytes in this block.

data area name Blank padded ASCII name of FORTRAN fixed data area.

daddr Starting address for this data.

data The initialization data.

00 * If the size of the data block is odd, there is one byte of 00 added to make the block an even number of bytes in size.

4.3.13 8C - FORTRAN Data Area Reference Block

```
                  +--------+--------+--------+--------+
byte -->      0   |   8C   |      size (3 bytes)      |
                  +--------+--------+--------+--------+
              4   |          data area name           |
                  |            (8 bytes)              |
                  +--------+--------+--------+--------+
             12   |          ref 1 (4 bytes)          |
                  +--------+--------+--------+--------+
             16   |          ref 2 (4 bytes)          |
                  +--------+--------+--------+--------+
                  |               . . .               |
                  +--------+--------+--------+--------+
                  | each reference consumes 4 bytes   |
                  +--------+--------+--------+--------+
                  |               . . .               |
                  +--------+--------+--------+--------+
          8+4*n   |          ref n (4 bytes)          |
                  +--------+--------+--------+--------+
```

How is this different from Common (or External) reference

8C — Hexadecimal 8C indicates this is a FORTRAN Data Area Reference Block.

size — Number of bytes in this block.

data area name — Blank padded ASCII name of FORTRAN fixed data area.

ref 1 — Location of first reference.

ref 2 — Location of first reference.

. . . — Location of other references.

ref n — Location of last reference.

4.3.14 8E - Quick Load Executable Block

```
             +--------+--------+--------+--------+
byte -->   0 |   8E   |      size (3 bytes)      |
             +--------+--------+--------+--------+
           4 |     start location (4 bytes)      |
             +--------+--------+--------+--------+
           8 |       data size (4 bytes)         |
             +--------+--------+--------+--------+
          12 | code block bytes (12..size-1) ... |
             +--------+--------+--------+--------+
```

8E — Hexadecimal 8E indicates this is a Quick-Load Executable Block.

size — Number of bytes in this block.

start location — Relative starting address of the code block.

data size — Total number of bytes in global common data areas.

code block — The absolute, self-relocatable code block for this program.

Implication?

4.3.15 8F - Executable Block Definition

```
                   +--------+--------+--------+--------+
byte -->       0   |   8F   |       size (3 bytes)     |
                   +--------+--------+--------+--------+
               4   | jump table address (4 bytes)      |
                   +--------+--------+--------+--------+
               8   |    jump table size (4 bytes)      |
                   +--------+--------+--------+--------+
              12   |      data size (4 bytes)          |
                   +--------+--------+--------+--------+
              16   |       num       |   00   |   00   |
                   +--------+--------+--------+--------+
              20   |   00   |   00   |   00   |   00   |
                   +--------+--------+--------+--------+
              24   |          size 1 (4 bytes)         |
                   +--------+--------+--------+--------+
              28   |          size 2 (4 bytes)         |
                   +--------+--------+--------+--------+
                   |               . . .               |
                   +--------+--------+--------+--------+
          24+n*4   |          size n (4 bytes)         |
                   +--------+--------+--------+--------+
          28+n*4   | jump table bytes (... size-1) ... |
                   +--------+--------+--------+--------+
```

8F   Hexadecimal 8F indicates this is an Executable Block Definition.

size   Number of bytes in this block.

jump table address
                  Absolute load address of jump table.

jump table size   Number of bytes in the jump table.

data size   Total number of bytes in global common data areas.

num   Number of FORTRAN Data Areas.

00 00 00 00 00 00
                  six bytes of zero filler.

size 1   Size of first FORTRAN Data Area.

size 2 Size of second FORTRAN Data Area.

. . . Sizes of other FORTRAN Data Areas.

size n Size of last FORTRAN Data Area.

jump table The jump table itself, including the executable code for the loader. For a further description, see the section on "Executable Block Details".

4.3.16 90 - Library Module Block

```
               +--------+--------+--------+--------+
byte -->     0 |   90   |      size (3 bytes)      |
               +--------+--------+--------+--------+
             4 |           module name             |
               |            (8 bytes)              |
               +--------+--------+--------+--------+
            12 |         msize (4 bytes)           |
               +--------+--------+--------+--------+
            16 |         caddr (4 bytes)           |
               +--------+--------+--------+--------+
            20 |         taddr (4 bytes)           |
               +--------+--------+--------+--------+
            24 |         tsize (4 bytes)           |
               +--------+--------+--------+--------+
            28 |  module count   |    module 1     |
               +--------+--------+--------+--------+
            32 |    module 2     |      . . .      |
               +--------+--------+--------+--------+
               |    module n-1   |    module n     |
               +--------+--------+--------+--------+
```

90 — Hexadecimal 90 indicates this is a Library Module Block.

size — Number of bytes in this block.

module name — Name of this module.

msize — Number of bytes of code in this module.

caddr — Disk address of module.

taddr — If non-zero, is the disk address of the text block. If zero, there is no text block.

tsize — Size of text block.

module count — Number of other modules that this module references.

module 1 — Number of the first module referenced.

module 2 — Number of the second module referenced.

. . . Numbers of other modules referenced.

module n Number of the last module referenced.

4.3.17 91 - Library Entry Block

```
               +--------+--------+--------+--------+
byte -->     0 |   91   |     size (3 bytes)       |
               +--------+--------+--------+--------+
             4 |            link name              |
               |            (8 bytes)              |
               +--------+--------+--------+--------+
            12 |     module      | . . . . . . . . .
               +--------+--------+--------+--------+
            14 |        address (4 bytes)          |
               +--------+--------+--------+--------+
```

?

4.3.18 92 - Unit Block

```
               +--------+--------+--------+--------+
byte -->    0  |   92   |       size (3 bytes)     |
               +--------+--------+--------+--------+
            4  |           unit  name              |
               |            (8 bytes)              |
               +--------+--------+--------+--------+
           12  |          caddr (4 bytes)          |
               +--------+--------+--------+--------+
           16  |          taddr (4 bytes)          |
               +--------+--------+--------+--------+
           20  |          tsize (4 bytes)          |
               +--------+--------+--------+--------+
           24  |          gsize (4 bytes)          |
               +--------+--------+--------+--------+
```

92 Hexadecimal 92 indicates that this is a Unit Block.

size Number of bytes in this block - always 00001C.

unit name Name of this unit.

caddr Disk address of module.

taddr Disk address of text block.

tsize Size of text block.

gsize Number of bytes of globals in this unit.

4.3.19 93 - FORTRAN Executable Data Area Reference Block

```
                    +--------+--------+--------+--------+
byte -->        0   |   93   |     size (3 bytes)       |
                    +--------+--------+--------+--------+
                4   |  area number    | . . . . . . . . .
                    +--------+--------+--------+--------+
                6   |          ref 1 (4 bytes)          |
                    +--------+--------+--------+--------+
               10   |          ref 2 (4 bytes)          |
                    +--------+--------+--------+--------+
                    |               . . .               |
                    +--------+--------+--------+--------+
                    | each reference consumes 4 bytes   |
                    +--------+--------+--------+--------+
                    |               . . .               |
                    +--------+--------+--------+--------+
           2+4*n    |          ref n (4 bytes)          |
                    +--------+--------+--------+--------+
```

93	Hexadecimal 93 indicates this is a FORTRAN Executable Data Area Reference Block.
size	Number of bytes in this block.
area number	Data area number.
ref 1	Address of first reference.
ref 2	Address of second reference.
. . .	Addresses of other references.
ref n	Address of last reference.

4.3.20 94 - FORTRAN Executable Data Area Initialization Block

```
                 +--------+--------+--------+--------+
byte -->       0 |   94   |     size (3 bytes)       |
                 +--------+--------+--------+--------+
               4 | data area number| . . . . . . . . .
                 +--------+--------+--------+--------+
               6 |          daddr (4 bytes)          |
                 +--------+--------+--------+--------+
              10 | initialization data . . . . . .   |
                 +--------+--------+--------+--------+
                 | . . . . . . . . . . . . . . . . . |
                 +--------+--------+--------+--------+
                 | . . . . . . . . . . . . . |  00   |
                 +--------+--------+--------+--------+
```

94 — Hexadecimal 94 indicates this is a FORTRAN Executable Data Area Initialization Block.

size — Number of bytes in this block.

data area number — Number of the FORTRAN Data Area.

daddr — Starting address for this data.

initialization data — The data to fill the block with.

00 — If the size of the initialization data is an odd number of bytes, a filler of 00 is appended to make it an even number of bytes.

4.4 Executable Block Details

This section describes the layout of an executable block. It includes details of the jump table and segment tables.

4.4.1 Layout of an Executable Block

```
                +--------+--------+--------+--------+
byte -->      0 |   8F   |      size (3 bytes)      |
                +--------+--------+--------+--------+
              4 |  Jump Table Address (4 bytes)     |
                +--------+--------+--------+--------+
              8 |   Jump Table Size (4 bytes)       |
                +--------+--------+--------+--------+
             12 |        Data Size (4 bytes)        |
                +--------+--------+--------+--------+
             16 |       Num       |   00   |   00   |
                +--------+--------+--------+--------+
             20 |   00   |   00   |   00   |   00   |
                +--------+--------+--------+--------+
             24 |          Size 1 (4 bytes)         |
                +--------+--------+--------+--------+
             28 |          Size 2 (4 bytes)         |
                +--------+--------+--------+--------+
                |               . . .               |
                +--------+--------+--------+--------+
         20+4*n |          Size n (4 bytes)         |
                +--------+--------+--------+--------+
         24+4*n | Jump Table (... size-1 bytes) ... |
                +--------+--------+--------+--------+
```

8F Hexadecimal 8F indicates this is an Executable Block Definition.

size Number of bytes in this block.

jump table address
Absolute load address of jump table.

jump table size Number of bytes in the jump table.

data size Total number of bytes in global common data areas.

num Number of FORTRAN Data Areas.

00 00 00 00 00 00
six bytes of zero filler.

size 1 Size of first FORTRAN Data Area.

size 2 Size of second FORTRAN Data Area.

. . . Sizes of other FORTRAN Data Areas.

size n Size of last FORTRAN Data Area.

jump table The jump table itself, including the executable code for the loader.

If any FORTRAN Executable Data Area Initialization Blocks are present, they must immediately follow the executable block.

4.4.2 Format of the Jump Table

```
                  +--------+--------+--------+--------+
A4 --> $$TOP      |    Number of Segments (2 bytes)    |
                  +--------+--------+--------+--------+
              +2  |   Main Segment Table (32 bytes)    |
                  +--------+--------+--------+--------+
             +34  |     Segment Table #2 (32 bytes)    |
                  |      . . . . . . . . . . .         |
                  |     Segment Table #n (32 bytes)    |
                  +--------+--------+--------+--------+
          2+n*32  |     Dummy Table #n+1 (4 bytes)     |
                  +--------+--------+--------+--------+
                  |   $_START Descriptor (10 bytes)    |
                  +--------+--------+--------+--------+
                  |     Segment #1 P#2 Descriptor      |
                  |      . . . . . . . . . . . .       |
                  |     'Segment #1 P#n Descriptor     |
                  +--------+--------+--------+--------+
                  |     Segment #2 P#1 Descriptor      |
                  |      . . . . . . . . . . . .       |  All segment
                  |     Segment #2 P#n Descriptor      |  descriptors
                  +--------+--------+--------+--------+  are 10 bytes.
                  |     Segment #3 P#1 Descriptor      |
                  +--------+--------+--------+--------+
                  |              . . .                 |
                  +--------+--------+--------+--------+
                  | Seg. #m P#n Descriptor (10 bytes)  |
                  +--------+--------+--------+--------+
             -20  |   Address of REMOVE1 (4 bytes)     |
                  +--------+--------+--------+--------+
             -16  |    Address of Buffer (4 bytes)     |
                  +--------+--------+--------+--------+
             -12  |  Address of Code File (4 bytes)    |
                  +--------+--------+--------+--------+
              -8  |   Active Segment List (4 bytes)    |
                  +--------+--------+--------+--------+
              -4  |     Address of $$TOP (4 bytes)     |
                  +--------+--------+--------+--------+
        $$LOADIT  |      Object-code neccessary to     |
                  |       load and execute a segment.  |
                  +--------+--------+--------+--------+
```

4.4.3 Layout of a Segment Table

A Segment Table consists of eight 32-bit values:

```
                  +--------+--------+--------+--------+
byte -->        0 |   Address of first descriptor     |
                  +--------+--------+--------+--------+
                4 |      File Address of Segment      |
                  +--------+--------+--------+--------+
                8 |       Size of code in bytes       |
                  +--------+--------+--------+--------+
               12 |     Actual Address in Memory      |
                  +--------+--------+--------+--------+
               16 |      Scratch Return Address       |
                  +--------+--------+--------+--------+
               20 |      Segment Reference Count      |
                  +--------+--------+--------+--------+
               24 |      Active Segment-list link     |
                  +--------+--------+--------+--------+
               28 |  . . .     Reserved        . . .  |
                  +--------+--------+--------+--------+
```

4.4.4 Layout of Descriptors

An entry-point-descriptor is in one of two states, depending whether its corresponding segement is in memory or not. The formats of a descriptor are:

```
When Segment not in memory:          When segment in memory:

+------------+-------------+         +------------+-------------+
| Relative offset of this  |         | Relative offset of this  |
+---                   ----+         +---                    ---+
|  entry in its segment.   |         |  entry in its segment.   |
+------------+-------------+         +------------+-------------+
|         JSR xxx.L        |         |         JMP xxx.L        |
+------------+-------------+         +------------+-------------+
|    Absolute address of   |         |    Absolute address of   |
+---                   ----+         +---                    ---+
|         $$LOADIT         |         |   procedure as loaded    |
+------------+-------------+         +------------+-------------+
```

4.5 Loading a Segment

A segment is loaded into memory when the first call to one of its procedures is executed. Such a call is always via a descriptor in the jump table.

The JSR to $$LOADIT executes the loader from its entry-point '$$LOADIT'. The loader is able to tell which segement to load by comparing the place from which it was called with the limits of the segment-table entries found in the first part of the jump table. The loader then performs the following actions:

1. The loader loads that segment.

2. Fixes up all the JSR's to JMP's, so that further calls upon that segment jump directly to the entry-point instead of calling the loader.

3. Saves the calling routine's return address in the segment entry.

4. Patches the return address on the stack to return through the anti-loader entry-point '$$REMOVE1'.

5. Jump to the procedure entry-point which caused this loader invocation in the first place.

Further calls to entry-points in the segment are thus only slowed by a single JMP instruction instead of a loader call. When the initial call to that segment eventually returns, it will pass through '$$REMOVE1', which removes that segment and reclaims the memory which that segment uses.

4.6 Running a Program

When a program is executed, the program called 'run' performs the following steps:

1. The file containing the executable program is opened,

2. It is checked to see if it is the correct format, for example, the first byte should be $8F_{16}$,

3. The jump table is loaded into the proper location in memory, and

4. A JSR to JT+Word(JT)*32+2 is executed.

The normal overlay procedure then takes control to overlay the main segment and begin execution at its starting address.

CORVUS CONCEPT

Technical Erratta Section

This is a preliminary list of files required to support the Corvus CONCEPT workstation. Files and file names may change between now and beta site distribution. Files marked with an * are required to boot the system.

L.E.F.

Volume: CCSYS, size = 2048 blocks

Operating system:

ASSIGN	9	data	*	Assign driver to device
CC.BOOTL	2	data		Local disk boot
CC.DISPAT	16	data	*	Dispatcher
CC.FILMGR	30	data	*	File manager
CC.HELP	7	data		System help program
CC.KERNEL	52	data	*	Operating system kernel
CC.SETPRT	15	data		Printer port set up
CC.SETUP	24	data	*	System initialization
CC.SYSMGR	16	data		System manager
CC.WNDMGR	23	data	*	Window manager
SHELL	12	data		System command processor
WRITEBOOT	5	data		Write boot blocks

Operating system drivers:

DRV.CONSOL	2	data	*	Console driver
DRV.DISPHZ	7	data	*	Horizontal display driver
DRV.DISPUD	7	data		Horizontal display driver
DRV.DISPVT	7	data	*	Vertical display driver
DRV.KYBD	6	data	*	Keyboard driver
DRV.PRNTR	3	data		Serial printer driver
DRV.SYSTRM	5	data	*	System terminal driver
DRV.TIMER	3	data	*	Timer (clock) driver

Character set files:

CSH.DEFAULT	4	data		Horizontal display character set
CSK.DEFAULT	2	data	*	Keyboard character set
CSU.DEFAULT	4	data	*	Horizontal display character set
CSU.ALTCHARSET	13	data		Alternate display character set
CSV.DEFAULT	3	data	*	Vertical display character set

Help data files:

H.DISPAT.TEXT	4	text		Dispatcher help text
H.FILMGR.TEXT	6	text		File manager help text
H.SYSMGR.TEXT	4	text		System manager help text
H.WNDMGR.TEXT	4	text		Window manager help text

System development files:

ASM68K	72	data		MC68000 assembler
CODE	89	data		Code file generator
DEBUG	12	data		Simple debugger
FORTRAN	185	data		Fortran compiler
LIBRARY	25	data		Library manager
LINKER	51	data		Code file linker
LOADER.IMAGE	1	data		

PASCAL	184	data	Pascal compiler
VSIPPP	20	data	Pascal program preprocessor
VSIXRF	30	data	Pascal program cross reference
FTNLIB.OBJ	217	data	Fortran support library
PASLIB.OBJ	53	data	Pascal support library

System support files:

DIAG.DATA	1	data	Disk diagnostic data
EDCH	28	data	Character set editor
MOVE	20	data	Disk block move program
ODIAG	45	data	OMNINET diagnostic program
SPOOL	28	data	Text file spool/despool program
ZAP	29	data	Disk block patch program

Application files:

CC.CPM	22	data	CP/M interpreter
CC.LGICLC1	94	data	LogiCalc
ED	199	data	EDWORD
EDINIT.TEXT	4	text	EDWORD support
LCMASK	9	data	LogiCalc support
SYSTEM.APPLECPM	26	data	CP/M support
ZED	96	data	EDWORD (Zentec version)

Demo files:

GRAPHICS	4C	data	Graphics demo
GDEMO	6	text	Graphics demo data
MEM	4	data	Plot memory (one line)
PLOTMEM	5	data	Plot memory
WDEMO	21	data	Window demo

File: cclib.doc.text
Date: 13-Apr-82

Corvus CONCEPT System Library

The /CCUTIL/CCLIB.OBJ library file contains support units and subroutines for the Corvus CONCEPT.

Units in the CCLIB library include:

CCdefn	- Corvus CONCEPT Definition Unit
CCclkIO	- Corvus CONCEPT Clock Processing Unit
CCcrtIO	- Corvus CONCEPT CRT Control Unit
CCdrvIO	- Corvus Disk Drive Support Unit
CCdrvU1	- Corvus Disk Drive Utilities Unit
CChexout	- Output Hex Characters Unit
CClblIO	- Corvus CONCEPT Label Processing Unit
CCpipes	- Corvus Disk Drive Pipes Unit
CCprtIO	- Corvus CONCEPT Printer I/O Unit
CCsema4	- Corvus Disk Drive Semaphore Unit
CCwndIO	- Corvus CONCEPT Window Processing Unit

Subroutines in the CCLIB library include:

OSactSlt - Get active slot function

```
          FUNCTION OSactSlt: integer;
```

OSactSrv - Get active server function

```
          FUNCTION OSactSrv: integer;
```

OSaltSlt - Get alternate slot function

```
          FUNCTION OSaltSlt: integer;
```

OSaltSrv - Get alternate server function

```
          FUNCTION OSaltSrv: integer;
```

OSsltTyp - Get device type for slot function

```
          FUNCTION OSsltType (slot: integer): slottype;
```

OSextCRT - Check for external CRT function

```
            FUNCTION OSextCRT: boolean;
```

OSmaxDev - Get maximum device number function

```
            FUNCTION OSmaxDev: integer;
```

OSdispDv - Get DISPLAY driver device number function

```
            FUNCTION OSdispDv: integer;
```

OSkybdDv - Get KYBD driver device number function

```
            FUNCTION OSkybdDv: integer;
```

OStimDv - Get TIMER driver device number function

```
            FUNCTION OStimDv: integer;
```

OSomniDv - Get OMNINET driver device number function

```
            FUNCTION OSomniDv: integer;
```

OSdcm2Dv - Get DTACOM2 driver device number function

```
            FUNCTION OSdcm2Dv: integer;
```

OSdcm1Dv - Get DTACOM1 driver device number function

```
            FUNCTION OSdcm1Dv: integer;
```

pOSuserID - Get Constellation user ID pointer

```
             FUNCTION pOSuserID: pointer;
```

pOScurWnd - Get current window record pointer

```
             FUNCTION pOScurWnd: pointer;
```

pOSsysWnd - Get system window record pointer

```
             FUNCTION pOSsysWnd (wndnbr: integer): pointer;
```

pOSdevNam - Get device name pointer

```
             FUNCTION pOSdevNam (untnbr: integer): pointer;
```

CCdefn Unit Interface

```
CONST
    MAXWINDOW   = 20;
    SysComPLoc  = $0180;
    LongStrMax  = 1030;
    MaxBytes    = 10000;

    {                                                           }
    { Corvus CONCEPT I/O Result Codes                           }
    {                                                           }

    IOEioreq = 03; { Invalid I/O request                        }

    IOEnotrn = 21; { Transporter not ready                      }
    IOEtimot = 22; { Timed out waiting for Omninet event        }
    IOEnobuf = 23; { Read without a valid write buffer          }

    IOEwndfn = 32; { Invalid window function                    }
    IOEwndbe = 33; { Window create boundary                     }
    IOEwndcs = 34; { Invalid character set                      }
    IOEwnddc = 35; { Delete current window                      }
    IOEwndds = 36; { Delete system window                       }
    IOEwndiw = 37; { Inactive window                            }
    IOEwndwr = 38; { Invalid window record                      }
    IOEwndwn = 39; { Invalid system window number               }

    IOEnodsp = 40; { Display driver not available               }
    IOEnokyb = 41; { Keyboard driver not available              }
    IOEnotim = 42; { Timer driver not available                 }
    IOEnoomn = 43; { OMNINET driver not available               }
    IOEnoprt = 44; { Printer driver not available               }

    IOEtblid = 50; { Invalid table entry ID                     }
    IOEtblfl = 51; { Table full                                 }
    IOEtbliu = 52; { Table entry in use                         }
    IOEkybte = 53; { Keyboard transmission error                }
    IOEuiopm = 54; { Invalid unit I/O parameter                 }
    IOEprmln = 55; { Invalid parameter block length             }
    IOEfnccd = 56; { Invalid function code                      }
    IOEclkmf = 57; { Clock (hardware) malfunction               }

TYPE
    Byte        = -128..127;
    String32    = STRING[32];
    pString32   = ^String32;
    String64    = STRING[64];
    pString64   = ^String64;
    String80    = STRING[80];
    pString80   = ^String80;
    Bytes       = ARRAY [0..9999] OF Byte;
    Words       = ARRAY [0..9999] OF INTEGER;
    pBytes      = ^Bytes;
    pWords      = ^Words;

    slottypes   = (nodrive,floppydrive,localdrive,omninet);
```

```
LongStr     = RECORD
                len: INTEGER;
                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;

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;
```

```
     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 }    attr2:  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 }    filll:  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}
{    2      35 }    rcdlen: byte;    {window description record length}
{ total     36 }    end;
```

CCclkIO Unit Interface

```
TYPE
    ClkStr2  = string[2];
    ClkStr10 = string[10];
    ClkStr40 = string[40];
    ClkPB    = record
               DayofWeek,Month,Day:              integer; { set by timer driver }
               Hour,Mins,Secs,Tenths,LeapYear: integer; { set by timer driver }
               end;

VAR
    ClkInfo:  ClkPB;     { clock parameter block }
    ClkDebug: boolean;   { debug flag  }
    ClkWD:    ClkStr10;  { day of week }
    ClkYr:    ClkStr10;  { year        }
    ClkMo:    ClkStr10;  { month       }
    ClkDy:    ClkStr2;   { day         }
    ClkHr:    ClkStr2;   { hour        }
    ClkMi:    ClkStr2;   { minute      }
    ClkSc:    ClkStr2;   { second      }
    ClkDate1: ClkStr40;  { date: "dy-mon-yr" format        }
    ClkDate2: ClkStr40;  { date: "month dy, year" format }
    ClkDate3: ClkStr40;  { date: "dy month year" format   }
    ClkTime1: ClkStr40;  { time: "hr:mi:sc" format         }
    ClkTime2: ClkStr40;  { time: "hr:mi am" format         }
    Year:     integer;   { set by unit ???     }

procedure ClkRead   (var CPB: ClkPB);
procedure ClkWrite  (CPB: ClkPB);
procedure ClkFormat (CPB: ClkPB);
procedure CCclkIOinit;
```

CCcrtIO Unit Interface

```
USES {$U CCLIB} CCdefn;

CONST
    CCcrtIOversion = 'n.n';
    YesEcho = TRUE;  NoEcho = FALSE;
    Shft    = TRUE;  NoShft = FALSE;
    Bsup    = TRUE;  NoBsup = FALSE;

TYPE
    CrtRdx     = (BinRdx,OctRdx,DecRdx,HexRdx);
    CrtStatus  = (Normal, Escape, Error);
    CrtCommand = (ErasEOS, ErasEOL, Up, Down, Right, Left, Leadin, EraseALL,
                  Tab, StartBeat, HeartBeat);

VAR
    Beep     : CHAR;
    CrtTpgm  : STRING[16];
    CrtTvrs  : STRING[16];
    CrtTcpy  : STRING[80];
    WndowLin : INTEGER;
    WndowCol : INTEGER;
    BeatCnt  : INTEGER;
    NumDef   : BOOLEAN;
    StrDef   : BOOLEAN;
    Shift    : BOOLEAN;
    Compress : BOOLEAN;
    TypeAhead: BOOLEAN;
    EchoCH   : BOOLEAN;
    RealCRT  : BOOLEAN;
    ExtCRT   : BOOLEAN;

FUNCTION  UpperCase  (ch: CHAR):          CHAR;
FUNCTION  GetNum     (VAR num:INTEGER):   CrtStatus;
FUNCTION  GetLongNum (VAR ln: LONGINT):   CrtStatus;
FUNCTION  GetString  (VAR buf:String80):  CrtStatus;
FUNCTION  GetByte:                        CHAR;
FUNCTION  CvStrInt   (VAR buf:String80):  INTEGER;
PROCEDURE CvIntStr   (num: INTEGER; VAR buf:String80; rdx:CrtRdx);
PROCEDURE CvLIntStr  (num: LONGINT; VAR buf:String80);
PROCEDURE CrtAction  (cmd: CrtCommand);
PROCEDURE CrtTitle   (txt: String80);
PROCEDURE CrtPrompt  (txt,opt: String80);
PROCEDURE CrtPause   (VAR ch: CHAR);
PROCEDURE GoToXY     (x,y: INTEGER);
PROCEDURE CCcrtIOinit;

{PROCEDURES/FUNCTIONS for compatibity}
PROCEDURE Crt       (cmd: CrtCommand);  {same as CrtAction}
```

CCdrvIO Unit Interface

```
USES {$U CCLIB} CCdefn;

CONST
    CCdrvioVersion = 'n.n';
    lowslot        = 1;
    highslot       = 5;

TYPE
    sevenbits = 0..127;
    eightbits = 0..255;
    aname     = PACKED ARRAY [1..4] OF CHAR;
    cdosbuf   = ARRAY [0..255] OF byte;

    trkaddr   = PACKED RECORD
                top3: 0..7;
                msb:  0..31;
                lsb:  0..255;
                END;

    voltabent = RECORD
                ftrk: trkaddr;
                ltrk: trkaddr;
                END;

    cbuffer   = ARRAY [0..127] OF trkaddr;

    volent    = RECORD
                ftype,
                lblk,
                fblk: INTEGER;
                vname: STRING[7];
                nfils,
                nblks: INTEGER;
                d2: PACKED ARRAY [0..7] OF CHAR;
                END;

    cvoldir   = RECORD
                ftype, lblk, fblk: INTEGER;
                name: STRING[7];
                nfils, nblks: INTEGER;
                fill: PACKED ARRAY [1..494] of CHAR;
                END;

    filent    = RECORD
                ftype,
                lblk,
                fblk: INTEGER;
                name: STRING[15];
                d2:   PACKED ARRAY [0..3] OF CHAR;
                END;

    cdir      = RECORD
                volu: volent;
                fil:  ARRAY [1..77] OF filent;
                END;
    userentry = PACKED RECORD
                name:       aname;
                password:   PACKED ARRAY[1..2] OF CHAR;
                bootvolume: eightbits;
                id:         sevenbits;
                pascaluser: BOOLEAN;
                END;

    ctable    = ARRAY [1..128] OF userentry;

    cdtyp     = (abuffer, avoldir, adir, atable, avbuf, adosbuf);

    cdbuf     = RECORD CASE cdtyp OF
                abuffer: (buffer: cbuffer);
                adir:    (dir:    cdir);
                atable:  (table:  ctable);
                adosbuf: (dosbuf: cdosbuf);
                avoldir: (voldir: cvoldir);
                END;

VAR
    drvCslot:    INTEGER;   {current slot number}
    drvPslot:    INTEGER;   {primary (boot) slot number}
    drvAslot:    INTEGER;   {alternate slot number}
    PrepFile:    FILE;
    PrepFID:     String32;

PROCEDURE cdsend (VAR st: SndRcvStr);
PROCEDURE cdrecv (VAR st: SndRcvStr);
PROCEDURE disksend (slot: INTEGER; VAR st: SndRcvStr);
PROCEDURE diskrecv (slot: INTEGER; VAR st: SndRcvStr);
FUNCTION  cdread  (VAR buf: cdbuf; len,drv,sct: INTEGER): INTEGER;
FUNCTION  cdwrite (VAR buf: cdbuf; len,drv,sct: INTEGER): INTEGER;
FUNCTION  PutPrep (VAR xcv: SndRcvStr; drv: INTEGER): INTEGER;
FUNCTION  UnPrep  (VAR xcv: SndRcvStr): INTEGER;
PROCEDURE CCdrvIOinit;
```

CCdrvU1 Unit Interface

```
USES
{$U CCLIB} CCdefn,
{$U CCLIB} CCdrvIO;

CONST
    CCdrvU1Version = 'n.n';
    DrMax          = 5;

TYPE
    DrRev       = (RevA,RevB,RevC);
    DrSizes     = (OldTenMB,FiveMB,TenMB,TwentyMB,FortyMB);
    VirDrInfo   = RECORD
                  Capacity: LONGINT;
                  END;
    PhysDrInfo = RECORD
                  spt,tpc,cpd: INTEGER;
                  Capacity: LONGINT;
                  DrSize: DrSizes;
                  DrType: DrRev;
                  PhysDr: BOOLEAN;  {true if physical drive, false for virtual}
                  END;
    VDrArray    = ARRAY [1..DrMax] OF VirDrInfo;
    PDrArray    = ARRAY [1..DrMax] OF PhysDrInfo;

VAR
    DrDebug:    BOOLEAN;
    DrTbuf:     CDBuf;    {general purpose I/O buffer}
    DrNumDrvs:  INTEGER;  {number of drives online}
    DrUserID:   INTEGER;  {current user ID}
                          { --- set by FindVol --- }
    DrVolDrv:   INTEGER;  {current volume disk drive}
    DrVolAddr:  INTEGER;  {current volume block address}
    DrVolIndex: INTEGER;  {current index into volume table}
                          {current disk volume table}
    DrVolTable: ARRAY [0..63] OF VolTabEnt;
                          { -------------------- }
    DrVirDrv: VDrArray;   {for call to CheckDrives}
    DrPhyDrv: PDrArray;   {ditto ...}

PROCEDURE DrvRd     (VAR Buf: CDBuf; Len,Drv,Sec: INTEGER);
PROCEDURE DrvWr     (VAR Buf: CDBuf; Len,Drv,Sec: INTEGER);
FUNCTION  GetAddr   (Trk: TrkAddr): INTEGER;
PROCEDURE ReadVT    (Drive,UserId: INTEGER);
FUNCTION  FindVol   (Mname: String32; Drive,UserID: INTEGER): INTEGER;
PROCEDURE CCdrvU1init;
```

CChexout Unit Interface

```
USES
{$U CCLIB} CCdefn;

PROCEDURE puthexbyte(b: BYTE);
PROCEDURE puthexword(w: INTEGER);
PROCEDURE puthexlong(l: LONGINT);
PROCEDURE dumphex(p: pBYTES; len: INTEGER);
PROCEDURE hexinit;
```

CClblIO Unit Interface

```
TYPE
    LblKeyStr = string[6];
    LblRtnStr = string[16];

PROCEDURE LblsInit;
PROCEDURE LblsOn;
FUNCTION  LblSet (KN: integer; LblStr: LblKeyStr;
                               RetStr: LblRtnStr): integer;
PROCEDURE CClblIOinit;
```

CCpipes Unit Interface

```
USES
{$U CCLIB} CCdefn;

CONST
    PipesVersion     = 'n.n'; {current version number}
    PnameLen         = 8;     {size of a pipe name}
    PblkLen          = 512;   {size of a pipe block}

    {pipe return codes ...}
    PipeOk           = 0;    {successful return code}
    PipeEmpty        = -8;   {tried to read an empty pipe}
    PipeNotOpen      = -9;   {pipe was not open for read or write}
    PipeFull         = -10;  {tried to  write to a full pipe}
    PipeOpErr        = -11;  {tried to open (for reading) an open pipe}
    PipeNotThere     = -12;  {pipe does not exist}
    PipeNoRoom       = -13;  {the pipe data structures are full, and there
                               is no room for new pipes at the moment...}
    PipeBadCmd       = -14;  {illegal command}
    PipesNotInitted = -15;  {pipes not initialized}
    {an error code less than -127 is a fatal disk error}

TYPE
    PNameStr   = STRING[PnameLen];
    PipeBlk    = RECORD CASE integer OF
                    1: (c: PACKED ARRAY [1..PblkLen] OF CHAR);
                    2: (b:        ARRAY [1..PblkLen] OF byte);
                 END;

VAR
    PipeCslot:   INTEGER; {current slot for pipe I/O}
    PipePslot:   INTEGER; {primary (boot) slot number}
    PipeAslot:   INTEGER; {alternate slot number}
    PipeDebug:   BOOLEAN;

FUNCTION pipestatus (VAR names,ptrs: PipeBlk): INTEGER;
FUNCTION pipeoprd   (pname: PNameStr): INTEGER;
FUNCTION pipeopwr   (pname: PNameStr): INTEGER;
FUNCTION pipeclrd   (npipe: INTEGER): INTEGER;
FUNCTION pipeclwr   (npipe: INTEGER): INTEGER;
FUNCTION pipepurge  (npipe: INTEGER): INTEGER;
FUNCTION piperead   (npipe: INTEGER; VAR info: PipeBlk): INTEGER;
FUNCTION pipewrite  (npipe,wlen: INTEGER; VAR info: PipeBlk): INTEGER;
FUNCTION pipesinit  (baddr,bsize: INTEGER): INTEGER;
PROCEDURE CCpipeinit;
```

CCprtIO Unit Interface

```
USES
{$U CCLIB} CCdefn;

CONST  PRT      = 6;            { unit # of /Printer }

       { baud rate codes }
       BAUD300  = 0;
       BAUD600  = 1;
       BAUD1200 = 2;
       BAUD2400 = 3;
       BAUD4800 = 4;            { default }
       BAUD9600 = 5;
       BAUD19200 = 6;

       { parity codes }
       PARDISABLED = 0;         { default }
       PARODD      = 1;
       PAREVEN     = 2;
       PARMARKXNR  = 3;
       PARSPACEXNR = 4;

       { datacom codes }
       PORT1       = 0;
       PORT2       = 1;         { default }

       { word size (charsize) codes }
       CHARSZ8     = 0;         { devault }
       CHARSZ7     = 1;

       { handshake codes }
       LINECTSINVERTED = 0;
       LINECTSNORMAL   = 1;
       LINEDSRINVERTED = 2;
       LINEDSRNORMAL   = 3;     { default }
       LINEDCDINVERTED = 4;
       LINEDCDNORMAL   = 5;
       XONXOFF         = 6;
       ENQACK          = 7;

VAR PrtAvail: boolean; { printer available (assigned) }

FUNCTION  PrtStatus    (var br,par,dc,chsz,hs: integer): integer;
FUNCTION  PrtFreeSpace (var freebytes: integer):         integer;
FUNCTION  PrtBaudRate  (baudrate: integer):              integer;
FUNCTION  PrtParity    (parity: integer):                integer;
FUNCTION  PrtDataCom   (port: integer):                  integer;
FUNCTION  PrtCharSize  (charsize: integer):              integer;
FUNCTION  PrtHandShake (protocol: integer):              integer;
PROCEDURE CCprtIOinit;
```

CCsema4 Unit Interface

```
USES
{$U CCLIB} CCdefn,
{$U CCLIB} CCdrvIO;

CONST
    Sema4version = 'n.n';

    { Return codes for the semaphore unit }

    SemWasSet = $80;    { the prior state of this semaphore was locked }
    SemNotSet = $00;    { prior state was unlocked }
    SemFull   = $FD;    { semaphore table is full (32 active semaphores) }
    SemDskErr = $FF;    { disk error during write thru }

    {  negative function return values indicate error conditions   }
    {  0 return means no error (and not set prior to operation)     }
    {  $80 (128) return means key set prior to operation            }

TYPE
    SemStr     = STRING[8];
    SemKeys    = PACKED ARRAY [1..8] OF CHAR;
    SemKeyList = RECORD CASE integer OF
                 1: (skey: ARRAY [1..32] OF SemKeys);
                 2: (sbyt: ARRAY [1..256] OF byte);
                 END;

FUNCTION SemLock   (key: SemStr): INTEGER;
FUNCTION SemUnlock (key: SemStr): INTEGER;
FUNCTION SemClear: INTEGER;
FUNCTION SemStatus (VAR kbuf: SemKeyList): INTEGER;
PROCEDURE CCSema4Init;
```

```
CCwndIO Unit Interface

USES
{$U CCLIB} CCdefn;

CONST
    GRAPHICS   = 2;     { attr2 flag values - add together }
    CURSORON   = 4;
    INVCURSOR  = 8;
    WRAPLINE   = 16;
    SCROLLOFF  = 32;
    CLEARPAGE  = 64;

    { values of wn for WinSystem }
    CURRPROCWIN = 1;     { current process window       }
    CMDWINDOW   =  2;    { cmd/msg window               }
    ROOTWINDOW  =  3;    { root user window             }

FUNCTION WinSystem (wn: integer):                                integer;
FUNCTION WinSelect (var WR: WndRcd):                             integer;
FUNCTION WinDelete (var WR: WndRcd):                             integer;
FUNCTION WinCreate (var WR: WndRcd; homex,homey,width,length: integer;
                    flags: byte):                                integer;
FUNCTION WinClear  (var WR: WndRcd):                             integer;
FUNCTION WinStatus (var homex,homey,width,length: integer):     integer;
PROCEDURE CCwndIOinit;
```