TI-99/4 AND 99/4A PERSONAL COMPUTER SYSTEM SOFTWARE

Comprehensive Specification

Consumer Group
Mail Station 5890
2301 N. University
Lubbock, Texas 79414

TEXAS INSTRUMENTS
INCORPORATED

Date: February 25, 1983
Version 1.0

TABLE of CONTENTS

LIST of TABLES

LIST of FIGURES

LIST of EXAMPLES

SECTION 1

INTRODUCTION

This document contains an overview of the TI-99/4 and TI-99/4A Personal Computer system software. It discusses details pertaining to the design and implementation of software for these products. For the most part the TI-99/4A Personal Computer system software is identical to that of the TI-99/4. This document will explain the additions, deletions and modifications which evolved the TI-99/4 system software into the TI-99/4A system software.

1.1 Purpose

The information provided herein is for use in maintenance of the system software for the 99/4 and 4A computers. It will serve as reference for the development of peripherals and for the design and development of future Personal Computer products. This manual is designed to serve as an end-user document as well, especially for programmers in assembly language. This document refers heavily to information in the other documents listed in section 2.0 and thus serves as a reference document.

1.2 Scope

This document is not intended to provide a detailed presentation of the hardware characteristics of the product although detail will be present where it is not available from other sources. More detail on the hardware characteristics can be found in the hardware specifications referred in section 2.0. This document discusses the design and features of the 99/4 and 4A console software and the software interface provided for software which may be plugged in on the Solid State Software and peripheral ports.

1.3 Terminology

CPU - Central Processing Unit

Communication Register Unit - An I/O technique for the TMS 9900 Microprocessor

CRU - Communication Register Unit

Device Service Routine - A routine that performs I/O operations to a device upon request from system and application software

DSR - Device Service Routine

GPL - Graphics Programming Language

Graphics Read-Only Memory - Memory-mapped I/O device which is a serial access ROM

GROM - Graphics Read-Only Memory

I/O - Input and/or Output

PAB - Peripheral Access Block

RAM - Random-Access Memory

Random-Access Memory - Randomly-accessed read/write memory

Read-Only Memory - Randomly-accessed readable but not writeable memory

ROM - Read-Only Memory

SSS - Solid-State Software

UDF - User Defined Function

VDP - Video Display Processor

Video Display Processor - Name of Texas Instruments TMS 9918 video generation chip

SECTION 2

APPLICABLE DOCUMENTS

TMS 9918A VDP Video Display Processor Data Manual
(Revised November 1982)

TMS 9919 Sound Generator Controller Specification
(Released 16 October 1979)

Texas Instruments Home Computer Technical Data
(Copyright 1980)

Home Computer Software Development System Programmer's Guide
(Revised 6 November 1979)

Home Computer BASIC Language Specification
(Revision 4.1 12 April 1979)

Texas Instruments Home Computer User's Reference Guide
(Learning Center Manual LCB-4491)

99/4 Home Computer I/O Bus Specification
(Electrical Specification: document number 1037185)

Texas Instruments Home Computer Editor/Assembler
(Copyright 1981)

SECTION 3

GENERAL DESCRIPTION

This section provides an overview of the system features provided in the hardware and software. The software exploitation of the hardware (e.g. ROM,RAM) is also discussed.

3.1 Hardware Description

These products use a plastic case with a number of plug-in ports for software modules and hardware extensions. Specifically two major ports are provided: one for an application cartridge and one for peripheral units. The application cartridge port allows a user to plug in a Solid State Command Module with up to 38K bytes of software. The peripheral port allows the addition of peripheral units (such as a disk or a RS-232 interface). With the Peripheral Expansion System, several pieces of hardware can reside in one box, therefore avoiding extra cables and clutter. Plugging the appropriate card into the system adds the desired function. The following functions are available:

Disk Drive Controller Card - Capability to add up to three disk drives, one within the P.E.S. itself.

Memory Expansion Card - Adds 32K bytes of RAM.

RS-232 Interface Card - Allows the Home Computer to transmit and receive data over phone lines (with the addition of the Telephone Coupler) and allows use of a printer.

P-Code Card - Adds UCSD Pascal language capability (requires Memory Expansion and Disk Drive for development).

Peripheral units will contain their own software Device Service Routine (DSR) within the unit but will not, in general, contain their own microprocessor. Other ports allow connection of the Handheld Controllers (joysticks) and up to two audio cassette recorders for data or BASIC program storage.

Speech capability is provided as a peripheral device. The software required to access the speech peripheral from the BASIC language is provided in application plug-in modules. Solid-state applications for the 99/4 and 4A which use speech must include software to access the speech hardware in the application software module.

3.1.1 Keyboards.

```
                                   ----------
                                   ¦ SHIFTED ¦
                                   ¦ NORMAL  ¦
                                   ----------

--------------------------------------------------------------------------------
¦  !  ¦  @  ¦  #  ¦  $  ¦  %  ¦  '  ¦  &  ¦  *  ¦  (  ¦  )  ¦
¦  1  ¦  2  ¦  3  ¦  4  ¦  5  ¦  6  ¦  7  ¦  8  ¦  9  ¦  0  ¦
--------------------------------------------------------------------------------
   ¦QUIT ¦BEGIN¦ UP  ¦REDO¦ERASE¦  >  ¦     ¦  -  ¦  +  ¦  "  ¦
   ¦  Q  ¦  W  ¦  E  ¦  R  ¦  T  ¦  Y  ¦  Ū  ¦  I  ¦  O  ¦  P  ¦
--------------------------------------------------------------------------------
¦     ¦ AID ¦ LEFT¦RIGHT¦ DEL ¦ INS ¦  <  ¦     ¦  /  ¦  =  ¦
¦SPACE¦  A  ¦  S  ¦  D  ¦  F  ¦  G  ¦  H  ¦  J  ¦  K  ¦  L  ¦
--------------------------------------------------------------------------------
   ¦     ¦ BACK¦ DOWN¦CLEAR¦PROCD¦  ?  ¦  :  ¦  ;  ¦  ,  ¦     ¦
   ¦SHIFT¦  Z  ¦  X  ¦  C  ¦  V  ¦  B  ¦  N  ¦  M  ¦  .  ¦ENTER¦
   -----------------------------------------------------------------------------
               ¦              SPACE              ¦
               -----------------------------------
```

Figure 3-1 TI-99/4 Keyboard

The 99/4 keyboard consists of 40 keys in a staggered array. The stagger of the keys is achieved by placing the keytops off center on the actual keys. The 99/4 keyboard does not include lower case letters or enough special characters to support international character set requirements.

```
                                                  ------------
                                                  ¦ FUNCTION ¦
                                                  ¦ SHIFTED  ¦
                                                  ¦ NORMAL   ¦
                                                  ------------

-------------------------------------------------------------------------
¦ DEL ¦ INS ¦ERASE¦CLEAR¦BEGIN¦PROCD¦ AID ¦REDO ¦BACK ¦     ¦QUIT ¦
¦  !  ¦  @  ¦  #  ¦  $  ¦  %  ¦  '  ¦  &  ¦  *  ¦  (  ¦  )  ¦  +  ¦
¦  1  ¦  2  ¦  3  ¦  4  ¦  5  ¦  6  ¦  7  ¦  8  ¦  9  ¦  0  ¦  =  ¦
-------------------------------------------------------------------------
   ¦     ¦  ~  ¦ UP  ¦  [  ¦  ]  ¦     ¦     ¦  ?  ¦  '  ¦  "  ¦     ¦
   ¦  Q  ¦  W  ¦  E  ¦  R  ¦  T  ¦  Y  ¦  U  ¦  I  ¦  O  ¦  P  ¦  -  ¦
   ¦  q  ¦  w  ¦  e  ¦  r  ¦  t  ¦  y  ¦  u  ¦  i  ¦  o  ¦  p  ¦  /  ¦
-------------------------------------------------------------------------
      ¦     ¦LEFT ¦RIGHT¦  {  ¦  }  ¦     ¦     ¦     ¦     ¦     ¦     ¦
      ¦  A  ¦  S  ¦  D  ¦  F  ¦  G  ¦  H  ¦  J  ¦  K  ¦  L  ¦  :  ¦ENTER¦
      ¦  a  ¦  s  ¦  d  ¦  f  ¦  g  ¦  h  ¦  j  ¦  k  ¦  l  ¦  ;  ¦     ¦
-------------------------------------------------------------------------
   ¦     ¦  \  ¦DOWN ¦  `  ¦     ¦     ¦     ¦     ¦     ¦     ¦     ¦
   ¦SHIFT¦  Z  ¦  X  ¦  C  ¦  V  ¦  B  ¦  N  ¦  M  ¦  <  ¦  >  ¦SHIFT¦
   ¦     ¦  z  ¦  x  ¦  c  ¦  v  ¦  b  ¦  n  ¦  m  ¦  ,  ¦  .  ¦     ¦
-------------------------------------------------------------------------
   ¦ALPHA¦     ¦                                            ¦     ¦
   ¦LOCK ¦CTRL ¦                  SPACE                     ¦FCTN ¦
-------------------------------------------------------------------------
```

Figure 3-2 TI-99/4A Keyboard

The 99/4A keyboard consists of 48 keys. There are three qualifier keys (shift, function, and control) making it possible to support lower case as well as special characters and communication characters (control characters). The above graphic approximately shows the appearance of the TI-99/4A keyboard. The words, DEL, INS, ERASE, etc., at the top of the keyboard are on an overlay and indicate the standard usage of the function keys. The ASCII characters shown on the lower portion of some letter keys are actually on the front of that key, i.e the question mark is on the front of the "I" key rather than on the top of the key. These "key front" characters are also function keys. The ALPHA LOCK key is an alternate action switch, i.e. a locking key.

Neither keyboard includes any hardware to scan the keys or provide interrupt signals to the 9900 microprocessor Keystrokes are not buffered in any way and the keyboards are not scanned by the software except on request of an application program. An exception to this is that the keyboards are scanned for the shift-Q (function = on 4A) key on every VDP vertical retrace interrupt which occurs 60 times a second.

3.1.2 Application Software Port.

The application module port provides plug-in capability for software provided by TI or third party authors in Solid State Command Modules. A Command Module will contain 1 to 5 TMS0430 "GROM" chips and up to 8K bytes of ROM in the form of 4764, 4732, 4716, etc. The ROM space may be expanded through the use of a paging scheme.

The port is designed such that when a command module is plugged in the machine is reset. This reset appears identical to the reset that occurs when the computer is powered up. This is done because the chips in the command module may cause spurious signals on the data and address busses when the module is plugged in. In particular the GROM chips in the module will not be synchronized to the same internal address as the GROMs built into the console. If the system software would read the GROM address at this time (the GROM address is read often) garbage would be obtained.

3.1.3 I/O Port.

The peripheral port provides all of the signal lines required to access the memory and CRU (Communication Register Unit) ports. The CRU is discussed in the following paragraphs and in the Texas Instruments Home Computer Editor/Assembler manual. The I/O port is fully described in the TI-99/4 Home Computer I/O Bus Specification.

3.1.3.1 Communications Register Unit.

The CRU, or Communications Register Unit, is a command-driven bit-addressable I/O interface which performs single and multiple bit programmed I/O. The CRU bus is used for peripheral enable and disable and for device control and data transfer to and from CRU mapped peripherals. All input consists of reading CRU line logic levels into memory, and all output consists of setting CRU output lines to bit values from a word or byte of memory. The CRU provides a maximum of 4096 input and 4096 output lines that may be individually selected by a 12-bit address in Workspace Register 12 (R12).

The CRU software base address is contained in R12. From the CRU software base address, the processor is able to determine the CRU hardware base address and the resulting CRU bit address. The CRU hardware base address is defined by bits 3 through 14 of the current R12 when CRU data transfer is performed. This 12-bit address is the base address for all CRU communications. Bits 0-2 and bit 15 of R12 are ignored for CRU address determination.

The CRU bit instructions use a displacement, in the range of -128 through +127, which is added to the CRU base address. An instruction can set, reset or test any bit in the CRU array or move data between the memory and CRU data fields. There are five instructions for communications with CRU lines. These instructions are:

SBO Set CRU Bit to One.
Sets a CRU output line to one.

SBZ Set CRU Bit to Zero.
Sets a CRU output line to zero.

TB Test CRU Bit.
Reads the digital input bit and sets the equal status bit (bit 2) to the value of the digital input bit.

LDCR Load Communications Register.
Transfers the number of bits (1-16) specified by the cnt field of the instruction to the CRU from the source operand. If Cnt=0, the number of bits transferred is 16. If the number of bits to be transferred is 1 to 8, the source address is a byte address. If the number of bits to be transferred is 9 to 16, the source address is a word address. It is important to note that R12 is unaltered by the LDCR instruction, even though the address is incremented as each successive bit is output.

STCR Store Communications Register.
Transfers the number of bits specified by the cnt field of the instruction from the CRU to the source operand. If Cnt=0, the number of bits transferred is 16. If the number of bits to be transferred is 1 to 8, the source address is a byte address. If the number of bits to be transferred is 9 to 16, the source address is a word address.

The CRU is used for system access to peripherals. There are 4K CRU bits, numbered >0000 through >0FFF. The CRU address loaded into R12 is twice the bit number. Thus, loading R12 with >1000 sets the base equal to CRU bit >800.

Of the available 4K of CRU bits, the first K, at addresses >0000 through >07FE, are used internally by the console. This includes the TMS9901 I/O chip, which address the keyboard, joysticks, cassette, etc. The second K, at addresses >0800 through >0FFE, are reserved for future use. The last 2K, at addresses >1000 through >1FFE, are reserved for the peripherals that are attached to the console port. A block of 128 CRU bits is assigned to each peripheral. A0 through A15 are the CPU address bus lines. CRU address 0 at A8 through A14 is the memory enable bit in each device address space. Setting the bit to 1 turns the device ROM/RAM on, and resetting it to 0 turns it off. This enables the address space from >4000 through >5FFF reserved for the peripheral ROM.

The CRU provides the most cost effective I/O for low and medium speed peripherals via the instruction driven serial data link. In applications where there are many I/O transfers of one or two bits, the CRU serial data link provides execution times that are better than for memory-mapped I/O, which always transfers 8 or 16 bits at a time. The CRU interface is simpler and therefore less expensive than memory-mapped I/O. The CRU interface requires fewer interface signals than the memory interface and can be expanded without affecting the memory system. In the majority of applications, where speed is not critical, the CRU I/O is superior to memory mapped I/O as a result of the powerful bit manipulation capability, flexible field lengths and simple bus structure.

3.1.4 CPU Memory Map.

This section essentially duplicates information contained in the Texas Instruments Home Computer Technical Data manual.

The 99/4 and 4A CPU memory map is designed to provide a great deal of expansion for future accessories and for compatibility with future products. The memory map is as follows:

```
0000 +------------+
     |            |
     |            |  ROM contained in the console
     |            |
     +------------+
2000 |            |
     |            |  CPU RAM expansion
     |            |
     +------------+
4000 |            |
     |            |  ROM in peripheral (mapped)
     |            |
     +------------+
6000 |            |
     |            |  ROM in application module (optional)
     |            |
     +------------+
8000 |            |
     |            |  CPU RAM and memory mapped devices
     |            |
     +------------+
A000 |            |
     |            |  CPU
     ~      :     ~   RAM
FFFF |            |    expansion
     +------------+
```

Figure 3-3 CPU Memory Map for 99/4 and 4A

The peripheral ROM is mapped into memory by selection through the CRU. The peripheral devices each have a unique CRU address. An access to that address maps the ROM for that device into the memory map. This addressing is further described in the Texas Instruments Home Computer Technical Data manual.

The block from 8000 to A000 contains the memory mapped devices and the 256 byte block of RAM on the CPU bus. Memory mapped devices (i.e. VDP, GROM, and SOUND chips) do not have fully decoded addresses. This causes certain locations in this block to be "unavailable". Actually the addresses of the chip memory mapped locations repeat in these lost areas. Only the recommended (primary) locations are explicitly identified below. Other locations are either unused or unavailable for use because of the partial decoding. All locations are one-byte data transfers except for CPU RAM. This area is subdivided as follows:

8300-83FF	256 byte CPU RAM
8400	Sound chip write data
8800	VDP read data
8802	VDP read status
8C00	VDP write data
8C02	VDP write address
9000	Speech read data
9400	Speech write data
9800	GROM page 0 read data
9802	GROM page 0 read address
9C00	GROM (GRAM) page 0 write data
9C02	GROM page 0 write address

Figure 3-4 Primary Memory Mapped Locations

The explanation of GROM pages and the expansion capability for GROM is given in section 3.1.6.

3.1.5 Video Display Processor.

The Video Display Processor (VDP) is accessable as a memory mapped device. Access to the VDP chip is through certain memory mapped locations. The uses of these locations are read status, write address, read data and write data. The memory addresses of the VDP access locations are listed in section 3.1.4. Details on the use of the VDP chip from 9900 assembly language is given in the "TMS 9918A VDP Video Display Processor Data Manual". The Graphics Programming Language Programmer's Guide describes the methods of accessing the VDP from that language.

3.1.6 GROM Memory Map.

The GROM memory map comprehends 3 GROM chips within the TI-99/4 and 4A console and up to 5 chips in a plug-in command module. Each GROM chip contains 6K bytes of data but resides on an 8K byte boundary. This leaves 2K "holes" in the address space which cannot be used. The mask program of each GROM chip contains the base address of the data in that chip. This corresponds to a chip number as illustrated in the figure below.

```
           +-  +------------------------------+
           !   ! chip 7     base E000         !
           !   +------------------------------+
application!   ! chip 6     base C000         !
   module  !   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           !   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
           !   ! chip 3     base 6000         !
           +-  +------------------------------+  -+
               ! chip 2     base 4000         !   !
               +------------------------------+   !
               ! chip 1     base 2000         !   ! built into
               +------------------------------+   !   console
               ! chip 0     base 0000         !   !
               +------------------------------+  -+
```

Figure 3-5 GROM Memory Map

The 5 chip limitation in an application module can be overcome if additional hardware can be provided in the application. The additional hardware would probably require a larger than standard application plastics or a separate box on the end of a cable. It may also require a power supply since the amount of power that can be drawn from the GROM port is very limited. The additional circuitry can decode the address lines for the ROM to provide several "pages" of GROM in the cartridge. These pages are then accessed by using different memory map locations for the device registers. The GROM register addresses listed in section 3.1.4 correspond to page 0 in this expanded scheme. The system software is designed to access 15 other pages as listed in the following table.

Table 3-1 Memory Mapping Addresses for Paged GROMs

PAGE	READ DATA	READ ADDRESS	WRITE DATA	WRITE ADDRESS
0	9800	9802	9C00	9C02
1	9804	9806	9C04	9C06
2	9808	980A	9C08	9C0A
3	980C	980E	9C0C	9C0E
4	9810	9812	9C10	9C12
5	9814	9816	9C14	9C16
6	9818	981A	9C18	9C1A
7	981C	981E	9C1C	9C1E
8	9820	9822	9C20	9C22
9	9824	9826	9C24	9C26
10	9828	982A	9C28	9C2A
11	982C	982E	9C2C	9C2E
12	9830	9832	9C30	9C32
13	9834	9836	9C34	9C36
14	9838	983A	9C38	9C3A
15	983C	983E	9C3C	9C3E

The three GROM chips built into the console are accessed by any of the application pages. This scheme is illustrated in the figure below.

```
               page 0          page 1          .... ...
           +-  +---------+     +---------+
           |   | chip 7  |     | chip 7  |
           |   +---------+     +---------+
application|   | chip 6  |     | chip 6  |
   module  |   ~~~~~~~~~~~     ~~~~~~~~~~~
           |   ~~~~~~~~~~~     ~~~~~~~~~~~
           |   | chip 3  |     | chip 3  |
           +-  +---------+-----+---------+----    -+
               |          chip 2             /     |
               +----------------------------/      |
               |          chip 1             /     | built into
               +----------------------------/      |   console
               |          chip 0             /     |
               +------------------------------    -+
```

Figure 3-6 GROM Access Pages

The initial program menu selection searches all 16 pages for valid programs and enters the correct page according to the user selection. Programs may branch between pages with the CALL subprogram feature describes on page H-6 of the GPL Programmer's Guide.

3.2 Software Description

The 99/4 and 4A software includes support for applications modules containing interpretive Graphics Programming Language (GPL) object code. GPL is a powerful assembly type language designed especially to provide easy access to the special graphics and sound features of the 99/4 and 4A hardware. Included within the 99/4 are a BASIC programming language and an Equation Calculator. The 99/4A includes a BASIC programming language, but not an Equation Calculator.

3.2.1 Features.

The console will support application modules containing programs written in GPL or BASIC (or a combination). The interpreter for GPL is contained in the system ROM. To reduce the size of application modules certain subroutines needed in the computer console have linkage provided for the use of applications modules. These subroutines are contained in the console GROM chips and include such features as trig functions, loading of standard character sets, all floating point math, etc.

A power-up program is contained in the console GROMs. This program initializes the system hardware and prompts the user for a menu selection of the desired application.

3.2.2 Supported Options.

Linkage to peripheral devices is provided in the computer console. The 99/4 and 4A peripherals contain software to service the device. The linkage routines allow applications to call a Device Service Routine (DSR) for a device by name and in a device independent manner. A detailed description of the requirements for an applicaton to request service from a peripheral is described in the File Management section (section 10).

The 99/4 and 4A console contains 16K bytes of RAM attached to the VDP chip. This RAM is not expandable beyond 16K bytes. The software is self-configuring with respect to the amount of VDP RAM so that a 99/4 derivative could be sold with less than 16K bytes of RAM. The amount of VDP RAM is placed in the GPL status block as part of the power-up sequence and must be tested by applications programs to ensure that enough RAM is available for the program to run. Peripheral devices may "steal" some of the VDP RAM at power-up time for use as buffers, etc. The peripherals will modify the location in the status block which specifies the memory size to reflect the amount of memory pre-empted.

3.2.3 ROM Usage.

The console ROMs contain the following software functions:

- GPL interpreter
- Radix 100 floating point package (+,-,* and /)
- Keyboard scan routine
- Subprogram/DSR search routine
- Low level audio cassette Device Service Routine (DSR)
- Interrupt processing including:
  - Auto-sound
  - Sprite motion
  - Interval timer
  - DSR interrupt
- BASIC interpreter program execution

3.2.4 GROM Usage.

The console GROMs contain the following software functions:

- GPL support routines including:
  - Subprogram and DSR linkage
  - Arithmetic and trigonometric functions
- System power-up and program selection
- High level Audio Cassette DSR
- Keyboard character (translation) tables
- User BASIC language editor and interpreter
- Equation Calculator interface to the BASIC interpreter
  (Not included on the 99/4A)

3.2.5 CPU RAM Usage.

During execution the BASIC interpreter has exclusive use of the first 140 bytes of the 256 bytes of on-board RAM. Otherwise, these bytes are free for programmer use. The remaining bytes are used by the GPL interpreter and various interrupt routines and peripheral devices. The CPU RAM has several uses as follows:

Free space for use by GPL applications
9900 workspace area (one workspace)
Partial workspace for interrupt handling
Work area for the GPL interpreter
GPL status block
Device Service Routine work area
CALL routine work area

These areas are described in detail in the GPL Programmer's Guide. The 9900 code uses only one workspace for all processing. Interrupts are only allowed when most workspace registers can be destroyed. The interrupt is taken into a second workspace where only registers 13 through 15 are used to save the interrupt status. Immediately after the interrupt is taken a LWPI instruction is executed to restore the context back to the one workspace. Another LWPI instruction is executed before the RTWP to return from the workspace context.

The unused space in the interrupt workspace is used by the GPL interpreter to save various information such as the last key pressed on the keyboard in order to debounce the keyboard. The two workspaces (one is partial) and this work area occupy 32 bytes of the 256 byte CPU RAM from address >83C0 to >83FF.

The GPL status block occupies location >836E through >837F. The use of these locations is documented in section 3.3.1 of the GPL Programmer's Guide. Locations >836E and >836F contain the Floating Point Stack pointer as described in Appendix K of the same manual. The following figure shows the CPU RAM segments dedicated to the GPL interpreter.

```
>8000 +---------------+
      |               |
>8310 |               |
      |               |
>8320 |               |
      |               |
>8330 |               |
      |               |
>8340 |               |
      +---------------+
>8350 |               |  >834A thru >836D will be destroyed
      |               |  when using peripherals.
>8360 |               |
      +---------------+
>8370 +---------------+
      | STATUS BLOCK  |
>8380 +---------------+  Default Subroutine Stack
      |               |
>8390 |               |
      |               |
>83A0 |     FREE      |  Default Data Stack
      |               |
>83B0 |               |
      |               |
>83C0 +---------------+
      |   INTERRUPT   |
>83D0 |   WORKSPACE   |
      |               |
>83E0 +---------------+
      |               |  R13 Address of GROM write address
>83F0 |  INTERPRETER  |  R14 System flags
      |   WORKSPACE   |  R15 Address of VDP write address
>83FF +---------------+
```

Figure 3-7 CPU RAM Memory Map

3.2.6 VDP RAM Usage.

The system utilization of VDP RAM can be closely controlled by a GPL application program. Many of the data structures for display on the screen are programmable through VDP registers and are described in the GPL Programmer's Guide. Certain data structures for sprites and the floating point roll out area are at fixed locations and may not be used by a particular application program. The following figure shows the VDP RAM segments dedicated to the GPL interpreter. The Sprite Descriptor Blocks are actually allocated from >000 through >7FF, but only use part of this space.

```
     >000 +---------------+
          :               :
          :   PATTERN     :
     >100 :               :
          :  NAME TABLE   :
          :               :
     >200 :  (768 bytes)  :
          :               :
          :               :
     >300 +---------------+  SPRITE ATTRIBUTE LIST (128 bytes)
          :               :
     >380 +---------------+  PATTERN COLOR TABLE (32 bytes)
     >400 +---------------+  FREE (96 bytes)
          +---------------+
          :               :
     >500 :    SPRITE     :
          :               :
          :  DESCRIPTOR   :
     >600 :               :
          :    BLOCKS     :
          :               :
     >700 :     (1K)      :
          :               :
     >780 +---------------+  SPRITE VELOCITY TABLE (128 bytes)
     >800 +---------------+
          :               :
          :   PATTERN     :
     >900 :               :
          :  GENERATOR    :
          ~               ~
          :    AREA       :
          :               :
     >F00 :    (2K)       :
          :               :  Assumes standard values
          :               :  in VDP registers.
    >1000 +---------------+
```

Figure 3-8 Default VDP RAM Memory Map under E/A

The VDP RAM is used as the primary memory for the BASIC interpreter, holding the BASIC program, symbol table, I/O buffer area and the string space, as well as the standard display, character and color table areas.

The following figure describes, in general, how the VDP RAM is partitioned for use by the BASIC interpreter. The Character Tables are actually allocated from >0000 through >07FF, but only use part of this space.

```
           +-----------------------------------------+   >0000
           |                                         |
           |                                         |
           |                 Screen                  |
           |                                         |
           |                                         |
           +-----------------------------------------+   >0300
           |        Color and Sprite Tables          |
CRNBUF  -->+-----------------------------------------+   >0320
           |                 Crunch                  |
           |                 Buffer                  |
CRNEND  -->+-----------------------------------------+   >03BE
           |           BASIC Temporaries             |
           |    and Interpreter Roll-out area        |
           +-----------------------------------------+   >0400
           |                                         |
           |            Character Tables             |
           |                                         |
STVSPT  -->+-----------------------------------------+   >0600
           |              Value Stack                |
 VSPTR  -->+-----------------------------------------+
           |                                         |
           |                                         |
           |                                         |
           |                                         |
STREND  -->+-----------------------------------------+
           |                                         |
           |              String Space               |
           |                                         |
 STRSP  -->+-----------------------------------------+
SYMPTR  -->|     Dynamic Symbol Table and PABs       |
SYMTAB  -->+-----------------------------------------+
           |                                         |
           |              Static Symbol              |
           |                  Table                  |
           |                                         |
  STLN  -->+-----------------------------------------+
           |            Line Number Table            |
  ENLN  -->+-----------------------------------------+
           |                                         |
           |            Crunched Program             |
           |                                         |
  @>70  -->+-----------------------------------------+   >3FFF
```

Figure 3-9 Console BASIC Memory Map

SECTION 4

TI-99/4A KEYBOARD SCAN ROUTINE

4.1 Introduction

The TI-99/4A has a 48-key typewriter style keyboard. In order to support this new device and maintain compatibility with existing software, a rather complex keyboard scan routine was written. The following sections explain the various aspects of this routine.

4.2 State of the Keyboard

The TI-99/4A keyboard has three possible states. They are:

1. TI-99/4 Emulator keyboard

2. Pascal keyboard

3. BASIC keyboard

Also, to maintain compatibility with existing software, the scan routine supports a split keyboard configuration.

The console software maintains an internal flag which determines the state of the keyboard. This flag can be controlled by any application by means of the keyboard number parameter in the GPL status block. The keyboard number parameter is in CPU RAM location >8374. The value 0 is used to scan the keyboard in whatever state it happens to be. The values 3, 4, and 5 are used to change keyboard states. The following paragraphs describe each state of the keyboard and the use of the keyboard number parameter to select that state.

4.2.1 TI-99/4 Emulator Keyboard.

The TI-99/4A powers up with the keyboard in this state. The keyboard number is set to zero. If an application needs to return to this state from some other state, the keyboard is scanned with keyboard number set to 3. Scanning keyboard number three sets the internal state flag to "99/4 Emulator", resets the keyboard number parameter to zero, and performs a keyboard scan in that state. Thereafter, the keyboard may be scanned with keyboard number set to 0.

In this state only 99/4 keyboard values are returned. There are three exceptions: [, \, and]. These characters were built into the 99/4 console but were not implemented on the keyboard. All other key codes are ignored and a no key condition is returned. For example, "CONTROL 1", which returns the value >B1, is not a legal 99/4 key. Therefore, the scan would return "No Key" for "CONTROL 1". Another result is that in this state the keyboard is "alpha-locked" regardless of the state of the ALPHA LOCK key.

4.2.2 Pascal Keyboard.

The keyboard is placed in this state by scanning with keyboard number set to 4. The internal state flag is set to "Pascal", the keyboard number is reset to 0, and the keyboard is scanned in this state. Thereafter, the keyboard may be scanned with keyboard number set to 0.

This is the state used by the UCSD P-System (Registered trademark of the Regents of the University of California). The complete ASCII range from 0 to >7F is returned. This includes standard control codes and the complete ASCII printable character set. In addition, codes from >81 through >8C, >8E and >8F, and >B0 through >C6 are returned for use as special function keys as follows:

>81 - AID
>82 - CLEAR
>83 - DELETE
>84 - INSERT
>85 - QUIT
>86 - REDO
>87 - ERASE
>88 - LEFT ARROW
>89 - RIGHT ARROW
>8A - DOWN ARROW
>8B - UP ARROW
>8C - PROCEED
>8E - BEGIN
>8F - BACK
>B0 through >C6 - application definable

4.2.3 BASIC Keyboard.

This state is selected by scanning with keyboard number set to 5. The internal state flag is set to "BASIC", the keyboard number is reset to 0, and the keyboard is scanned in this state. Thereafter, the keyboard may be scanned with keyboard number set to 0.

This state is used by BASIC, Terminal Emulator II and text editors. The differences between the keyboards are as follows:

1. ASCII control codes have the most significant bit set, i.e. >00 through >1F on the Pascal keyboard become >80 through >9F on the BASIC keyboard.

2. The defined application special function keys do not have the most significant bit set, i.e. >81 through >8C, >8E and >8F on the Pascal keyboard become >01 through >0C, >0E and >0F on the BASIC keyboard.

4.3 Keyboard Levels

Within each of the keyboard states there are five levels controlled by the four modifier keys: CONTROL, FUNCTION, SHIFT, and ALPHA LOCK. The precedence of the modifier keys is the order listed. For example, if CONTROL and any other modifier are down simultaneously, CONTROL will take precedence. The following three tables show the key codes returned by each of the states of the keyboard on each of the levels. The ENTER and SPACE keys are not included because they return >0D and >20 respectively, regardless of the state or level of the keyboard. NK is used to indicate the "no key" condition. This consists of returning a >FF key code and a reset condition bit.

Table 4-1 TI-99/4 Emulator Keyboard - Keycodes

KEY	UNMODIFIED	ALPHA LOCK	CONTROL	FUNCTION	SHIFT
1	>31	A	NK	>03	>21
2	>32	L	NK	>04	>40
3	>33	P	NK	>07	>23
4	>34	H	NK	>02	>24
5	>35	A	NK	>0E	>25
6	>36		NK	>0C	>5E
7	>37	L	NK	>01	>26
8	>38	O	NK	>06	>2A
		C			
9	>39	K	NK	>0F	>28
0	>30		NK	NK	>29
=	>3D	H	NK	>05	>2B
Q	>51	A	NK	NK	>51
W	>57	S	NK	NK	>57
E	>45		NK	>0B	>45
R	>52	N	NK	>5B	>52
T	>54	O	NK	>5D	>54
Y	>59	E	NK	NK	>59
U	>55	F	NK	>5F	>55
I	>49	F	NK	>3F	>49
O	>4F	E	NK	>27	>4F
P	>50	C	NK	>22	>50
/	>2F	T	NK	NK	>2D
A	>41		NK	NK	>41
S	>53	I	NK	>08	>53
		N			
D	>44		NK	>09	>44
F	>46	T	NK	NK	>46
G	>47	H	NK	NK	>47
H	>48	I	NK	NK	>48
J	>4A	S	NK	NK	>4A
K	>4B		NK	NK	>4B
L	>4C	S	NK	NK	>4C
;	>3B	T	NK	NK	>3A
		A			
Z	>5A	T	NK	>5C	>5A
X	>58	E	NK	>0A	>58
C	>43		NK	NK	>43
V	>56		NK	NK	>56
B	>42		NK	NK	>42
N	>4E		NK	NK	>4E
M	>4D		NK	NK	>4D
,	>2C		NK	NK	>3C
.	>2E		NK	NK	>3E

Table 4-2 Pascal Keyboard - Keycodes

KEY	UNMODIFIED	ALPHA LOCK	CONTROL	FUNCTION	SHIFT
1	>31	>31	>B1	>83	>21
2	>32	>32	>B2	>84	>40
3	>33	>33	>B3	>87	>23
4	>34	>34	>B4	>82	>24
5	>35	>35	>B5	>8E	>25
6	>36	>36	>B6	>8C	>5E
7	>37	>37	>B7	>81	>26
8	>38	>38	>1E	>86	>2A
9	>39	>39	>1F	>8F	>28
0	>30	>30	>B0	>BC	>29
=	>3D	>3D	>1D	>85	>2B
Q	>71	>51	>11	>C5 or >B9	>51
W	>77	>57	>17	>7E	>57
E	>65	>45	>05	>8B	>45
R	>72	>52	>12	>5B	>52
T	>74	>54	>14	>5D	>54
Y	>79	>59	>19	>C6	>59
U	>75	>55	>15	>5F	>55
I	>69	>49	>09	>3F	>49
O	>6F	>4F	>0F	>27	>4F
P	>70	>50	>10	>22	>50
/	>2F	>2F	>BB	>BA	>2D
A	>61	>41	>01	>7C	>41
S	>73	>53	>13	>88	>53
D	>64	>44	>04	>89	>44
F	>66	>46	>06	>7B	>46
G	>67	>47	>07	>7D	>47
H	>68	>48	>08	>BF	>48
J	>6A	>4A	>0A	>C0	>4A
K	>6B	>4B	>0B	>C1	>4B
L	>6C	>4C	>0C	>C2	>4C
;	>3B	>3B	>1C	>BD	>3A
Z	>7A	>5A	>1A	>5C	>5A
X	>78	>58	>18	>8A	>58
C	>63	>43	>03	>60	>43
V	>76	>56	>16	>7F	>56
B	>62	>42	>02	>BE	>42
N	>6E	>4E	>0E	>C4	>4E
M	>6D	>4D	>0D	>C3	>4D
,	>2C	>2C	>00	>B8	>3C
.	>2E	>2E	>1B	>B9	>3E

Table 4-3 BASIC Keyboard - Keycodes

KEY	UNMODIFIED	ALPHA LOCK	CONTROL	FUNCTION	SHIFT
1	>31	>31	>B1	>03	>21
2	>32	>32	>B2	>04	>40
3	>33	>33	>B3	>07	>23
4	>34	>34	>B4	>02	>24
5	>35	>35	>B5	>0E	>25
6	>36	>36	>B6	>0C	>5E
7	>37	>37	>B7	>01	>26
8	>38	>38	>9E	>06	>2A
9	>39	>39	>9F	>0F	>28
0	>30	>30	>B0	>BC	>29
=	>3D	>3D	>9D	>05	>2B
Q	>71	>51	>91	>C5 or >B9	>51
W	>77	>57	>97	>7E	>57
E	>65	>45	>85	>0B	>45
R	>72	>52	>92	>5B	>52
T	>74	>54	>94	>5D	>54
Y	>79	>59	>99	>C6	>59
U	>75	>55	>95	>5F	>55
I	>69	>49	>89	>3F	>49
O	>6F	>4F	>8F	>27	>4F
P	>70	>50	>90	>22	>50
/	>2F	>2F	>BB	>BA	>2D
A	>61	>41	>81	>7C	>41
S	>73	>53	>93	>08	>53
D	>64	>44	>84	>09	>44
F	>66	>46	>86	>7B	>46
G	>67	>47	>87	>7D	>47
H	>68	>48	>88	>BF	>48
J	>6A	>4A	>8A	>C0	>4A
K	>6B	>4B	>8B	>C1	>4B
L	>6C	>4C	>8C	>C2	>4C
;	>3B	>3B	>9C	>BD	>3A
Z	>7A	>5A	>9A	>5C	>5A
X	>78	>58	>98	>0A	>58
C	>63	>43	>83	>60	>43
V	>76	>56	>96	>7F	>56
B	>62	>42	>82	>BE	>42
N	>6E	>4E	>8E	>C4	>4E
M	>6D	>4D	>8D	>C3	>4D
,	>2C	>2C	>80	>B8	>3C
.	>2E	>2E	>9B	>B9	>3E

4.4 Returned Information and Debounce

The 99/4A keyboard scan returns three bytes of information. The first of these is modification of the keyboard number parameter (CPU RAM >8374). When keyboard numbers 3, 4, and 5 are scanned, the 99/4A resets the keyboard number to 0. This fact can be used by an application to determine whether it is running on a 99/4 or a 99/4A. The second piece of information is the keycode of the current key. >FF is used to indicate "No Key Down". The keycode value is returned in CPU RAM >8375. The final bit of information is the key status which is returned in the GPL STATUS byte (CPU RAM >837C). If the key is a new key, bit 5 (GPL condition bit) is set. If the key is an old key, this bit is reset.

Debounce is done by key station, not by keycode value. As a result when a key is held down, changes in the level of the keyboard (with the exception of ALPHA LOCK) do not affect the keycode returned. Thus if "CONTROL 1" is held down, >B1 is repeatedly returned. If just the CONTROL key is released, the scan will still return >B1. If the FUNCTION key is now pressed, the scan will still return >B1. In other words >B1 will be returned until either the "1" key is released or a key with higher precedence is pressed.

The 99/4A keyboard routine stops scanning as soon as it finds a key. As a result the keyboard has a fixed hierarchy, and the key stations assume the following order of precedence:

Z Q A 1 O P ; / B T G 5 6 Y H N V
R F 4 7 U J M C E D 3 8 I K , X W
S 2 9 0 L . ENTER SPACE =

Thus in the previous example where the "1" key is being held down, if the "Z", "Q", or "A" key is now pressed, they will take precedence over the "1" key, and the returned code will change.

In addition a time-delay debounce was added to the 99/4A keyboard routine. Each time a new key is found, the routine delays for 10 milliseconds. This delay is to avoid multiple entries from one keypush, an existing problem on the 99/4. Since keys bounce when released as well as when pressed, the routine also performs the delay when it finds a "No Key" condition immediately following a "Key Down" condition.

4.5 Split Keyboard and Joystick Scans

These two features are combined as on the 99/4. Executing the routine with keyboard number set to 1 scans joystick unit 1 and the left side split keyboard. If the keyboard number is set to 2, the routine scans joystick unit 2 and the right side split keyboard. Scanning the split keyboard does not affect the internal flag which determines the state of the console keyboard. In this respect the console keyboard and the split keyboard are separate devices. However, because the 99/4 did not treat them as separate devices and because of compatibility issues with existing software, they are not treated as separate devices with respect to debounce.

4.5.1 Returned Information.

Scanning a joystick/split-keyboard returns the following information:

>8375 - Returned keycode; the fire button on the joystick unit takes precedence over any key on the split keyboard

>8376 - Y joystick parameter;
4 = pushed forward, 0 = centered, -4 = pulled back

>8377 - X joystick parameter;
4 = pushed right, 0 = centered, -4 = pushed left

>837C - Status of returned keycode

4.5.2 Split Keyboard 1.

The following table shows the key station assignments for split keyboard 1 and the corresponding return values.

Table 4-4 Split Keyboard 1 - Keycodes

KEY	KEYCODE	KEY	KEYCODE	KEY	KEYCODE	KEY	KEYCODE
1	19	Q	18	A	1	Z	15
2	7	W	4	S	2	X	0
3	8	E	5	D	3	C	14
4	9	R	6	F	12	V	13
5	10	T	11	G	17	B	16

All other keys return a "No Key" condition.

The fire button on joystick unit one is logically identical to the "Q" key with one exception, the fire button takes precedence over all the other keys, the "Q" key does not.

4.5.3 Split Keyboard 2.

The following table shows the key station assignments for split keyboard 2 and the corresponding return values.

Table 4-5 Split Keyboard 2 - Keycodes

KEY	KEYCODE	KEY	KEYCODE	KEY	KEYCODE	KEY	KEYCODE
6	19	Y	18	H	1	N	15
7	7	U	4	J	2	M	0
8	8	I	5	K	3	,	14
9	9	O	6	L	12	.	13
0	10	P	11	;	17	/	16

All other keys return a "No Key" condition.

The fire button on joystick unit two is logically identical to the "Y" key with one exception, the fire button takes precedence over all the other keys, the "Y" key does not.

4.6 Assembly Language Interface

The 99/4A keyboard routine may be used from 9900 assembly language. Entry is accomplished by a BL @>OE instruction. The following inputs are required:

1. The proper keyboard number in CPU RAM >8374.
2. CPU RAM >83D4 must contain the current value of VDP register 1.
3. The GPL workspace (>83E0) must be used when a BL @>OE instruction is executed.
4. All interrupts must be disabled before a BL @>OE instruction. (LWPI 0)
5. CPU RAM >8373 must contain a one-byte pointer into CPU RAM, i.e. the least significant byte of a CPU RAM address. This pointer is the GPL subroutine stack pointer. The scan routine pushes the current GROM address (2 bytes) on this stack, then pops it off after

the scan. The stack is a pre-incrementing one.

6. CPU RAM >83C6 through >83CA must contain the information stored there by the previous keyboard scan. This is keyboard state and debounce information and must be maintained.

7. CPU RAM >8314 must contain a copy of VDP RAM Register 1. It is used to turn on the screen when a key is pressed.

Execution of the scan routine modifies the following CPU RAM locations:

1. The word located two bytes higher than the address indicated by the pointer in CPU RAM >8373
2. >8374 - Keyboard number; reset from 3, 4, or 5 to 0
3. >8375 - Returned keycode
4. >8376 - Y joystick parameter; modified when keyboard number 1 or 2 is scanned
5. >8377 - X joystick parameter; modified when keyboard number 1 or 2 is scanned
6. >837C - Key status; cleared for old keys; >20 for new keys
7. >83C6 through >83CA - Debounce and internal flags; these values must be maintained between scans for the routine to function properly
8. >83D6 and >83D7 - Screen timeout; cleared after each new key
9. >83D8 and >83D9 - Save return address during scan routine
10. R0 through R7, R11, and R12 of the GPL workspace

SECTION 5

CONSOLE SOFTWARE

This section provides an overview of the system features provided by the software contained in the 99/4 computer console. These features are provided to support application program modules and access to peripheral units from application modules.

5.1 System Power-up Sequence

Most of the system power-up initialization is written in the interpretive GPL language. When the system is powered up the level 0 interrupt is taken. This interrupt vector is at address 0 and loads the workspace to >83E0. After loading R13 with the GROM read address the GPL interpreter starts interpreting GPL codes at GROM location >20. The GPL code performs the rest of the initialization as follows:

- Loads R15 with the VDP write address (>8C02)
- Loads R14 with status flags
- Clears the sound list indicator used for auto-sound (location >83CE)
- Turns off the speech chip (when attached) and the sound generators
- Initializes the two GPL stacks (subroutine and data) in the status block
- Initializes the VDP registers to default values
- Zeros much of CPU RAM: >8300 - >8371, >8382 - >83BF, >83C2 - >83C9 (or >83C2 - >83D7)
- Enables the audio gate so that the cassette data will be heard on the monitor speaker
- Enables the VDP 60 Hertz interrupt
- Enables the external interrupt
- Enables audio cassette motors (to the On state)
- Issues a beep to signal powered up
- Determines the VDP memory size and sets the VDP register bit accordingly
- Clears the first 4K bytes of VDP RAM
- Loads the default color and character tables
- Initializes all keyboards by scanning them
- Displays the power-up screen (screen turned off)
- Calls possible Power-up screen modification routine in cartridge. This is done for foreign languages.

Calls power-up routines in ROM and GROM (see a description of the GROM/ROM header). Power up routines may modify the VDP RAM size placed in location >8370.
Turns on VDP screen (it is turned off during initialization)
Waits for a key on the console keyboard, then beeps
Builds a list of the available programs including looking for a library. Only GROM is searched.
Displays the program menu screen (screen turned off)
Calls possible menu screen modification routine in cartridge. This is done for foreign languages.
Turns the screen on
Waits for user menu selection
Branches to starting address of program

If the console only contains GROM 0 (as a future product) and no cartridge is inserted then the menu screen will display "INSERT CARTRIDGE" instead of the menu. This cannot happen in the 99/4A because BASIC is always present.

A user selected program must always be a GPL program or must at least be started in GPL. The GPL program may initiate assembly language by the XML instruction or may initiate BASIC from GROM as described in the Home Computer Software Development System Programmer's Guide. Some of the 99/4As can initiate a ROM program without a GROM.

5.2 <u>GPL Application Support</u>

The GPL application support consists of the GPL object code interpreter and the callable console GPL subroutines. In addition, the peripheral support described in section 5.4.3 allows access to peripherals from an application program. These applications are developed on a 990/4, 990/10 or 990/12 minicomputer using the development aids described in Appendix C. Development of GPL application programs by the end user of the computer is not supported and there is no plan to do so for the general user.

5.2.1 <u>GPL Interpreter</u>.

The GPL object interpreter consists of 9900 assembly routines to interpret the object code generated by the GPL assembler. A floating point arithmetic package is also included. This package consists of assembly language routines accessable by the GPL XML instruction and GPL routines accessable by the CALL statement. The floating point routines use an 8-byte radix 100 floating point representation which provides at least 13 digits of significance.

5.2.2 Support Subroutines.

Certain GPL subroutines in the 99/4 console which could be useful to application programs are made accessable by a branch table at a fixed location in the console GROMs. The use of this branch table provides a fixed address to enter the routines even if the console code changes. The routines are discussed in Appendix K of the GPL Programmer's Guide. The complete list of routines is given below:

Table 5-1 GPL Routines

Address	Name	Use	Ref
10	LINK	Link to subprograms and DSRs	GPL App H
12	RETN	Return from subprogram or DSR	GPL App H
14	CNS	Convert floating point to ASCII	GPL App K
16	CHR1	Load 8 dot high character set	GPL App H
18	CHR2	Load 7 dot high character set	GPL App H
1A	BWARN	Warning message from BASIC subprogram in GPL	
1C	BERR	Error message from BASIC subprogram in GPL	
1E	BEXEC	Begin execution of GROM BASIC program	
20	PWRUP	Restart system (used at power-up)	
22	INT	Greatest integer of a floating point number	GPL App K
24	PWR	Exponentiation	GPL App K
26	SQR	Square root	GPL App K
28	EXP	Inverse natural logarithm	GPL App K
2A	LOG	Natural logarithm	GPL App K
2C	COS	Cosine	GPL App K
2E	SIN	Sine	GPL App K
30	TAN	Tangent	GPL App K
32	ATN	Arctangent	GPL App K
34	TON1	Good prompt tone	GPL App H
36	TON2	Bad prompt tone	GPL App H
4A	CHR3	Load 7 dot high small capitals (99/4A only)	

Certain 9900 routines are accessable from GPL through the XML instruction. Many of these routines are also accessable as subroutines (with R11 as the link) to the 9900 code in the console. However they are not available to external 9900 code (in peripherals or Command Modules) because their addresses are not fixed. The XML routines use the following CPU RAM locations:

FAC is CPU RAM >834A (8 bytes).
ARG is CPU RAM >835C (8 bytes).
STATUS is CPU RAM >837C.

When errors occur during the execution of floating point (base 100) routines, they are indicated by a non-zero value being placed in CPU RAM location FAC+10 (CPU RAM >8354). If an error has occured, the user program is then responsible for clearing this error flag location. The XML routines are as follows:

Table 5-2 XML Routines

XML Number	Name	Use	Ref
00	unused	unused	
01			GPL App K
02			GPL App K
03			GPL App K
04			GPL App K
05			GPL App K
06	FADD	Floating Point Addition FAC <- ARG + FAC	GPL App K
07	FSUB	Floating Point Subtraction FAC <- ARG - FAC	GPL App K
08	FMUL	Floating Point Multiplication FAC <- ARG * FAC	GPL App K
09	FDIV	Floating Point Division FAC <- ARG / FAC	GPL App K
0A	FCOMP	Floating Point Compare STATUS <- ARG, FAC	GPL App K
0B	SADD	Value Stack Addition STACK -> ARG ; then FADD	GPL App K
0C	SSUB	Value Stack Subtraction STACK -> ARG ; then FSUB	GPL App K
0D	SMUL	Value Stack Multiplication STACK -> ARG ; then FMUL	GPL App K
0E	SDIV	Value Stack Division STACK -> ARG ; then FDIV	GPL App K
0F	SCOMP	Value Stack Compare STACK -> ARG ; then FCOMP	GPL App K
10	CSN	Convert String to F.P. Number FAC <- FAC	GPL App K
11			GPL App K
12	CFI	Convert F.P. to Integer: Rounded conversion of f.p. to integer. FAC <- FAC	GPL App K
13			
14			

More information on the general use of XML instructions is given in the Texas Instruments Home Computer Editor/Assembler manual.

5.2.3 Application Configuration.

Application programs may only be contained in GROM. The user selects an application program from the system menu. A program is placed in the system menu by its reference in a GROM header. The GROM header is defined in Appendix H of the GPL Programmer's Guide. An example of a GROM header for an application program is given below.

```
        GROM 3
        ORG  0
        DATA >AA             HEADER IDENTIFIER
        DATA 0               VERSION NUMBER (NOT USED)
        DATA 1               NUMBER OF PROGRAMS (NOT USED)
        DATA 0               NOT USED (RESERVED)
        DATA #0              ADDRESS OF POWER-UP HEADER (NONE HERE)
        DATA #PROG1          ADDRESS OF APPLICATION PROGRAM HEADER
        DATA #0              ADDRESS OF DSR HEADER (NONE HERE)
        DATA #0              ADDRESS OF SUBPROGRAM HEADER (NONE HERE)
        DATA #0              ADDRESS OF INTERRUPT LINK (NONE IN GROM)
        DATA #0              UNUSED
*
PROG1   DATA #0              LINK TO NEXT HEADER (NONE)
        DATA #START          ENTRY POINT
        DATA 19              LENGTH OF PROGRAM NAME
        DATA :APPLICATION PROGRAM:   PROGRAM NAME
```

Example 5-1 Application Program GROM Header

The GROM header may be placed at the beginning (address 0) of any GROM chip. When the system starts an application program the system memory is initialized to mostly zeros as described in Appendix H of the GPL Programmer's Guide.

A link editor is not provided for GPL to resolve references between separately assembled modules. A technique that has proven useful is to place a branch table at the beginning of a module for those routines which are referenced by other assemblies. In this way the addresses of external routines remain fixed although the actual routine address may move within the separate assembly.

5.3 BASIC Interpreter

The BASIC interpreter is a GPL application program which is built into the 99/4 console. To provide sufficient speed many of the core execution routines are written in 9900 assembly language. Linkage to these routines is through system defined XML instructions. All of the edit and symbol table generation portions of the BASIC interpreter are written in GPL. Much more detail of the design of the BASIC interpreter can be found in the TI 99/4 Home Computer BASIC Interpreter Design Specification.

5.4 Peripheral Support

The 99/4 system provides for peripheral Device Service Routines to be contained in peripheral ROM or in system or application cartridge GROM. ROM DSRs are written in 9900 assembly language while GROM DSRs are written in GPL. A DSR may contain a power-up routine if that is required for the device. An interrupt entry is only provided for ROM (9900 assembly language) DSRs.

5.4.1 General Concepts.

The three entry points (power-up, I/O call, interrupt) are defined in the ROM/GROM header. The same header structure is used in either ROM or GROM although some fields are not used in one or the other. For example the interrupt field in a GROM header and the application program field in a ROM header are not used. The ROM header must be placed at the beginning (address >6000) of a ROM DSR.

```
        DATA >AA00          HEADER IDENTIFIER, VERSION NUMBER
        DATA 0              NOT USED
        DATA PWR            ADDRESS OF POWER-UP HEADER
        DATA 0              NOT USED
        DATA DSR            ADDRESS OF DSR HEADER
        DATA 0              ADDRESS OF SUBPROGRAM HEADER (NONE HERE)
        DATA INT            ADDRESS OF INTERRUPT LINK
        DATA 0              UNUSED
*
PWR     DATA 0              LINK TO NEXT HEADER (NONE)
        DATA PWRUP          POWER UP ENTRY
*
DSR     DATA 0              LINK TO NEXT HEADER (NONE)
        DATA START          ENTRY POINT
        BYTE 19             LENGTH OF PROGRAM NAME
        TEXT 'APPLICATION PROGRAM'  PROGRAM NAME
*
INT     DATA 0              LINK TO NEXT HEADER (NONE)
        DATA INTENT         INTERRUPT ENTRY
```

Example 5-2 Application Program ROM Header

The power-up and interrupt headers do not have name entries in them.

5.4.2 Power-up.

The power-up entry of every DSR is executed before the power-up "press any key" screen is placed on the monitor. ROM power-up routines are executed before GROM power-up routines. Power up routines may use CPU RAM locations >8304 through >83BF. Note that these locations cannot all be used in the other DSR entries. ROM power-up routines may use registers R0 through R10 at will. Other registers contain the following:

R11	return address
R12	CRU base address
R13	GROM read data address
R14	system flags
R15	VDP write address address

R12 must contain the same value on the return to monitor as it contained on entry to the DSR. DSRs will often change the address in R12 but since peripherals reside on >100 boundaries R12 can be restored with an ANDI R12,>FF00 instruction.

5.4.3 I/O Calls.

During an I/O call the DSR may use CPU RAM locations >834A through >836D. A ROM DSR may also use >83EA through >83EF if it does not enable interrupts. A ROM device service routine may use registers R0 through R10 at will. Other registers contain the following:

R11	return address
R12	CRU base address
R13	GROM read data address
R14	system flags
R15	VDP write address address

R12 must contain the same value on the return to monitor as it contained on entry to the DSR. DSRs will often change the address in R12 but since peripherals reside on >100 boundaries R12 can be restored with an ANDI R12,>FF00 instruction.

A GPL I/O routine returns to the monitor with a CALL RETN (see section 5.2.2). Note that a CALL must be used and not a branch. A ROM (9900) I/O routine must return with the following code.

```
INCT R11            or            B @2(R11)
RT
```

5.4.4 Interrupts.

When an external interrupt occurs the system interrupt handler enters the interrupt entry of each peripheral device. Each device service routine must determine if its device requires service and handle it. An interrupt routine returns to the monitor with a 9900 "RT" instruction.

ROM interrupt routines may use R0 through R7 and R10 at will. R8 may be used but must be cleared (set to zero) before exiting the interrupt routine. R9 cannot be destroyed. The use of R11 through R15 is as described above. Other areas of CPU RAM are not available for use in an interrupt routine.

SECTION 6

TI-99/4A BASIC

6.1 Functional Changes

The following is a list of the functional differences between 99/4 BASIC and 99/4A BASIC:

1. 99/4A BASIC uses the "BASIC" version of the keyboard described in section 4.2.3. The state of the keyboard can be changed by CALL KEY. However, the state reverts to "BASIC" whenever BASIC returns to the command level, e.g. at program termination or on a breakpoint.

2. Both upper and lower case character definitions are initialized. Since BASIC uses dynamic allocation for the "graphics" characters, this means that 99/4A BASIC powers up with 256 fewer bytes of free memory than 99/4 BASIC does.

3. Input in the edit mode or in response to a program INPUT statement now includes an auto-repeat feature. A key which is held down for a second will begin to repeat at a rate of 12 characters per second.

4. The "Equation Calculator" was removed.

5. Failure of an OLD command no longer necessarily destroys the program previously in memory. Since the current program may or may not be partially overwritten before the "OLD" failure, a warning message is displayed.

6. The cursor was changed to a solid rectangular box.

6.2 Bug Fixes

The following is a list of bugs in 99/4 BASIC which were fixed in 99/4A BASIC:

1. Line numbers listed to a peripheral device are printed properly at a record boundary.
2. The POS function will now find substrings beyond position 127.
3. 255 byte records are now properly blank-filled.
4. String expressions in CALL SOUND do not crash the system.
5. User-defined numeric functions with string parameters will return values near zero.
6. Programs with breakpoints may be safely edited.
7. Editing a program followed by performing an imperative command will not destroy the program.
8. Multi-line insertions and deletions do not garbage the margins.
9. Dimension wrap-around at 65536 has been fixed.
10. Garbage collection lockup when using files has been fixed.
11. Illegal long constants in INPUT, READ, etc. generate error messages rather than crashing system.
12. Illegal uses of UDFs cause error messages rather than destroying the program.

SECTION 7

GPL INTERPRETER MODIFICATIONS

Three known bugs in the GPL interpreter were fixed. The bugs and the fixes are described below.

7.1 CRU IN

The CRU IN portion of the GPL I/O instruction has never worked. The interpreter uses the information provided by the GPL instruction to create the appropriate 9900 CRU instruction. The 9900 instruction is then executed in a register. Because an increment-by-two instruction was used in place of an increment instruction, the original code was not creating a legal CRU IN instruction. Changing the increment-by-two to an increment fixed this bug.

7.2 CASE

The GPL CASE instruction has caused problems in some applications programs. The problem is one of timing. The original code repeatedly performs two successive GROM reads with no time delay between them. Therefore, some GROMs which pass virtually every other test may fail to execute a particular CASE statement because of this timing problem. To fix this problem, a NOP was added between the GROM reads.

7.3 FETCH

The orginal GPL FETCH instruction would not fetch data into VDP RAM. The source of this problem was a register conflict. A register containing certain flags was being used to save another value during execution of the FETCH instruction. This bug was fixed by using a free register to save the necessary value during FETCH execution.

SECTION 8

VDP INTERRUPT HANDLING

One problem with the TI-99/4 is the inflexibility of the VDP interrupt routine. In an effort to make some previously impossible applications doable, the VDP interrupt routine was modified. This routine performs six basic functions:

1. Sprite motion
2. Auto-sound
3. System reset key (Shift-Q)
4. Screen timeout
5. GPL timer increment
6. Storing VDP status in GPL status block.

On the TI-99/4A the first three of these functions are optional. The last three are not. Also the ability exists to execute an additional interrupt routine. The execution of each phase of the resident interrupt routine depends upon a bit flag in CPU RAM >83C2. If the most significant nybble of this byte is 0, all phases of the interrupt routine will execute. Setting any of the bits in this nybble will disable some part of the interrupt handler as follows:

bit 7 - disable sprite motion, auto-sound, and system reset key

bit 6 - disable sprite motion

bit 5 - disable auto-sound

bit 4 - disable system reset key

Bit 7 is the most significant bit.

After the routine has handled sprite motion, auto-sound, and checked for the system reset key, the timing functions are performed and the VDP status is stored. The interrupt handler then checks whether there is an external routine to be executed. The existence of an external routine is determined by the word value at CPU RAM >83C4. If this location is zero, there is no external routine and the interrupt handling is complete. If this location contains a non-zero value, it is assumed to be a pointer to another interrupt routine and control is passed to that location.

At this point in the processing the workspace pointer is pointing to the GPL workspace (>83E0). If the routine is to use this workspace, the values in certain registers must be preserved as follows:

1. If the routine is operating in the GPL environment, i.e. control will return to the GPL interpreter, the values in registers 13, 14, and 15 must be preserved.

2. If the routine is operating in the BASIC or Extended BASIC environment, i.e. control will return to either the BASIC interpreter or the GPL interpreter running BASIC, the values of registers 9, 10, 13, 14, and 15 must be preserved. Also, register 8 (>83F0) of the GPL workspace must be cleared regardless of whether or not that workspace is used.

An external routine may conclude by returning to the console routine with the GPL workspace active. At this time register 8 is cleared and control is returned to the point at which the interrupt occured. If preferred, an external routine may return directly to the point at which the interrupt occured by loading the interrupt workspace pointer (>83C0) and performing an RTWP instruction.

SECTION 9

OTHER MODIFICATIONS

9.1 BREAK KEY ROUTINE

One of the major problems in converting from the 99/4 keyboard to the 99/4A keyboard was the location of the BASIC "BREAK" key. On the 99/4 "Shift-C" was used both as the BASIC "BREAK" key and the RS232 "ABORT" key. Unfortunately the BASIC interpreter, Extended BASIC interpreter, and RS232 directly scan the keyboard lines to check for this key. The problem arises from the fact that "Function-4", instead of "Shift-C", is the "BREAK" key on the 99/4A. Therefore, some rearranging had to be done on the 99/4A keyboard to insure that "Function-4" was located in the keyboard matrix at the same position as "Shift-C" on the 99/4 keyboard.

To avoid this problem in the future, a "Check for BREAK key" routine was added to the console software. This routine is executed via a BL @>20 instruction. This routine only modifies the value of register 12. If the BREAK key is down, it returns to the caller with the equal bit set. Otherwise it returns with the equal bit reset. All future software which needs to check for this key must use this routine rather than testing the keyboard lines directly.

9.3 New Character Definitions

The 99/4A keyboard has the capability of generating the full ASCII character range from 0 to >7F. To further enhance this capability, definitions for the complete ASCII printable character set were added to the console. The previous 5x6 uppercase character set was increased in size to 5x7 and lowercase definitions were added. Due to software compatibility issues the console will still power-up with the 6x8 uppercase character set and the character loading routines are as follows:

1. Console routine >16 loads the 6x8 character set which includes characters >20 through >5F

2. Console routine >18 loads the 5x7 uppercase character set which includes characters >20 through >5F

3. Console routine >4A loads the 5x7 lowercase character set which includes characters >60 through >7E

These routines are executed by storing the starting VDP RAM address in CPU RAM location >834A and performing a GPL CALL instruction to the appropriate routine. The following pages show the definitions of the new 5x7 character set.

Table 9-1 5X7 Character Set

```
code=32 " "    code=33 "!"    code=34 """    code=35 "#"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1              1    X         1  X   X       1  X   X
2              2    X         2  X   X       2  X   X
3              3    X         3  X   X       3X X X X X
4              4    X         4              4  X   X
5              5    X         5              5X X X X X
6              6              6              6  X   X
7              7    X         7              7  X   X

code=36 "$"    code=37 "%"    code=38 "&"    code=39 "'"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1  X X X       1X X           1  X           1      X
2X   X   X     2X X     X     2X   X         2      X
3X   X         3      X       3X   X         3    X
4  X X X       4    X         4  X           4
5    X   X     5  X           5X   X   X     5
6X   X   X     6X     X X     6X     X       6
7  X X X       7      X X     7  X X   X     7

code=40 "("    code=41 ")"    code=42 "*"    code=43 "+"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1      X       1  X           1              1
2    X         2    X         2  X   X       2    X
3  X           3      X       3    X         3    X
4  X           4      X       4X X X X X     4X X X X X
5  X           5      X       5    X         5    X
6    X         6    X         6  X   X       6    X
7      X       7  X           7              7

code=44 ","    code=45 "-"    code=46 "."    code=47 "/"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1              1              1              1
2              2              2              2        X
3              3              3              3      X
4              4X X X X X     4              4    X
5  X X         5              5              5  X
6    X         6              6  X X         6X
7  X           7              7  X X         7
```

```
code=48 "0"    code=49 "1"    code=50 "2"    code=51 "3"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1  X X X       1    X         1  X X X       1  X X X
2X       X     2  X X         2X       X     2X       X
3X       X     3    X         3        X     3        X
4X       X     4    X         4      X       4    X X
5X       X     5    X         5    X         5        X
6X       X     6    X         6  X           6X       X
7  X X X       7  X X X       7X X X X X     7  X X X

code=52 "4"    code=53 "5"    code=54 "6"    code=55 "7"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1      X       1X X X X X     1    X X       1X X X X X
2    X X       2X             2  X           2        X
3  X   X       3X X X X       3X             3      X
4X     X       4        X     4X X X X       4    X
5X X X X X     5        X     5X       X     5  X
6      X       6X       X     6X       X     6  X
7      X       7  X X X       7  X X X       7  X

code=56 "8"    code=57 "9"    code=58 ":"    code=59 ";"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1  X X X       1  X X X       1              1
2X       X     2X       X     2  X X         2  X X
3X       X     3X       X     3  X X         3  X X
4  X X X       4  X X X X     4              4
5X       X     5        X     5  X X         5  X X
6X       X     6      X       6  X X         6    X
7  X X X       7  X X         7              7  X

code=60 "<"    code=61 "="    code=62 ">"    code=63 "?"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1      X       1              1  X           1  X X X
2    X         2              2    X         2X       X
3  X           3X X X X X     3      X       3        X
4X             4              4        X     4      X
5  X           5X X X X X     5      X       5    X
6    X         6              6    X         6
7      X       7              7  X           7    X
```

```
code=64 "@"     code=65 "A"     code=66 "B"     code=67 "C"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1  X X X        1  X X X        1X X X X        1  X X X
2X       X      2X       X      2  X     X      2X       X
3X   X X X      3X       X      3  X     X      3X
4X   X   X      4X X X X X      4  X X X        4X
5X   X X X      5X       X      5  X     X      5X
6X              6X       X      6  X     X      6X       X
7  X X X        7X       X      7X X X X        7  X X X

code=68 "D"     code=69 "E"     code=70 "F"     code=71 "G"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1X X X X        1X X X X X      1X X X X X      1  X X X X
2  X     X      2X              2X              2X
3  X     X      3X              3X              3X
4  X     X      4X X X X        4X X X X        4X   X X X
5  X     X      5X              5X              5X       X
6  X     X      6X              6X              6X       X
7X X X X        7X X X X X      7X              7  X X X

code=72 "H"     code=73 "I"     code=74 "J"     code=75 "K"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1X       X      1  X X X        1        X      1X       X
2X       X      2    X          2        X      2X     X
3X       X      3    X          3        X      3X   X
4X X X X X      4    X          4        X      4X X
5X       X      5    X          5        X      5X   X
6X       X      6    X          6X       X      6X     X
7X       X      7  X X X        7  X X X        7X       X

code=76 "L"     code=77 "M"     code=78 "N"     code=79 "O"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1X              1X       X      1X       X      1X X X X X
2X              2X X   X X      2X X     X      2X       X
3X              3X   X   X      3X X     X      3X       X
4X              4X   X   X      4X   X   X      4X       X
5X              5X       X      5X     X X      5X       X
6X              6X       X      6X     X X      6X       X
7X X X X X      7X       X      7X       X      7X X X X X
```

```
code=80 "P"     code=81 "Q"     code=82 "R"     code=83 "S"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1X X X X        1  X X X        1X X X X        1  X X X
2X       X      2X       X      2X       X      2X       X
3X       X      3X       X      3X       X      2X
4X X X X        4X       X      4X X X X        4  X X X
5X              5X   X   X      5X   X          5        X
6X              6X     X        6X     X        6X       X
7X              7  X X   X      7X       X      7  X X X

code=84 "T"     code=85 "U"     code=86 "V"     code=87 "W"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1X X X X X      1X       X      1X       X      1X       X
2    X          2X       X      2X       X      2X       X
3    X          3X       X      3X       X      3X       X
4    X          4X       X      4  X   X        4X   X   X
5    X          5X       X      5  X   X        5X   X   X
6    X          6X       X      6    X          6X   X   X
7    X          7  X X X        7    X          7  X   X

code=88 "X"     code=89 "Y"     code=90 "Z"     code=91 "["
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1X       X      1X       X      1X X X X X      1  X X X
2X       X      2X       X      2        X      2  X
3  X   X        3  X   X        3      X        3  X
4    X          4    X          4    X          4  X
5  X   X        5    X          5  X            5  X
6X       X      6    X          6X              6  X
7X       X      7    X          7X X X X X      7  X X X

code=92 "\"     code=93 "]"     code=94 " "     code=95 "_"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1               1  X X X        1               1
2X              2      X        2    X          2
3  X            3      X        3  X   X        3
4    X          4      X        4X       X      4
5      X        5      X        5               5
6        X      6      X        6               6
7               7  X X X        7               7X X X X X
```

```
code=96 "`"    code=97 "a"    code=98 "b"    code=99 "c"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1              1              1              1
2  X           2              2              2
3    X         3  X X X       3X X X X       3  X X X X
4      X       4X       X     4  X     X     4X
5              5X X X X X     5  X X X       5X
6              6X       X     6  X     X     6X
7              7X       X     7X X X X       7  X X X X

code=100"d"    code=101"e"    code=102"f"    code=103"g"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1              1              1              1
2              2              2              2
3X X X X       3X X X X X     3X X X X X     3  X X X X
4  X     X     4X             4X             4X
5  X     X     5X X X X       5X X X X       5X   X X X
6  X     X     6X             6X             6X       X
7X X X X       7X X X X X     7X             7  X X X

code=104"h"    code=105"i"    code=106"j"    code=107"k"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1              1              1              1
2              2              2              2
3X       X     3  X X X       3    X X X     3  X     X
4X       X     4    X         4      X       4  X   X
5X X X X X     5    X         5      X       5  X X
6X       X     6    X         6X     X       6  X   X
7X       X     7  X X X       7  X X         7  X     X

code=108"l"    code=109"m"    code=110"n"    code=111"o"
 1 2 3 4 5      1 2 3 4 5      1 2 3 4 5      1 2 3 4 5
1              1              1              1
2              2              2              2
3X             3X       X     3X       X     3X X X X X
4X             4X X   X X     4X X     X     4X       X
5X             5X   X   X     5X   X   X     5X       X
6X             6X       X     6X     X X     6X       X
7X X X X X     7X       X     7X       X     7X X X X X
```

```
code=112"p"     code=113"q"     code=114"r"     code=115"s"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1               1               1               1
2               2               2               2
3X X X X        3  X X X        3X X X X        3  X X X X
4X       X      4X       X      4X       X      4X
5X X X X        5X   X   X      5X X X X        5  X X X
6X              6X     X        6X     X        6        X
7X              7  X X   X      7X       X      7X X X X

code=116"t"     code=117"u"     code=118"v"     code=119"w"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1               1               1               1
2               2               2               2
3X X X X X      3X       X      3X       X      3X       X
4    X          4X       X      4X       X      4X       X
5    X          5X       X      5  X   X        5X   X   X
6    X          6X       X      6  X   X        6X   X   X
7    X          7  X X X        7    X          7  X   X

code=120"x"     code=121"y"     code=122"z"     code=123"{"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1               1               1               1    X X
2               2               2               2  X
3X       X      3X       X      3X X X X X      3  X
4  X   X        4  X   X        4      X        4X
5    X          5    X          5    X          5  X
6  X   X        6    X          6  X            6  X
7X       X      7    X          7X X X X X      7    X X

code=124"|"     code=125"}"     code=126"~"
 1 2 3 4 5       1 2 3 4 5       1 2 3 4 5
1    X          1  X X          1
2    X          2      X        2  X
3    X          3      X        3X   X   X
4               4        X      4      X
5    X          5      X        5
6    X          6      X        6
7    X          7  X X          7
```

SECTION 10

FILE MANAGEMENT

10.1 Introduction

This section contains a complete description for the File Management System of the TI-99/4 and 4A Home Computer. This description includes information about File structures, File I/O, the Peripheral Access Block, I/O Opcodes and DSR operations.

10.2 I/O Handling

The approach used in the TI-99/4 Home Computer File Management System, is that all devices, with the exception of the screen and the keyboard, look the same to an application program. Therefore, when peripherals are added to the computer, the BASIC Interpreter is not affected. Only the peripheral drivers (Device Service Routines or DSRs) have to be added to the software. Physical limitations (like reading data from a thermal printer which is clearly impossible) are determined in the DSR and are returned to the application program as an error condition.

The 99/4 and 4A personal computer File Management System supports two kinds of file organization:

1. Sequential files

2. Relative record files

Both file types use the same supervisor call mechanism, in order to insure a high degree of device independence. Hardware devices (such as a line printer) are accessed as sequential files, except that no file name is appended to the device name.

10.2.1 File Organization and Use.

The following paragraphs discuss the file organization and use for the two file types.

10.2.1.1 Sequential Files.

The records in sequential files can only be read from, or written to, in sequential order. This is appropriate for printers, modems, cassettes and some disk-based files. The records in sequential files can be either fixed or variable length.

10.2.1.2 Relative Record Files.

The records in relative files can be read from, or written to, in either sequential order or in random order. Relative record files are also called random access files, since records may be accessed in an arbitrary order. Therefore relative record files can only be supported on diskettes. The records in relative files are always fixed length. This enables the system to compute the actual location of any logical record relative to the beginning of a file.

The records within a relative record file are addressed by a unique record number. To access record X, the value X has to be placed in the appropriate field of the I/O Peripheral Access Block. Each record has a number from zero up to one less than the number of records in the file.

Records in a relative record file can also be accessed in a sequential way, by specifying the first record in the sequence only. The Supervisor then automatically updates the record number each time after a record has been read.

10.2.2 File Characteristics.

10.2.2.1 Terminology.

A file consists of a collection of data groupings called logical records. These records do not necessarily correspond with the physical divisions of the data in the file (like a sector on a disk). Thus, there are two types of records:

1. Logical records - The data grouping of a file as seen by an application program.

2. Physical records - The buffers physically transferred between memory and medium.

File I/O is done on a logical record basis. Manipulation of physical records is handled by the DSR.

When a file is created, its characteristics must be defined. Most of these characteristics cannot be changed later in the file's existence. The logical record size is an attribute which must be specified. For relative record files this size must be exact. For sequential files the specification indicates an upper limit for the record size. In case a zero is specified for either filetype, the DSR must select a default for the record size. The physical record size for any medium is specified within the DSR and is implementation dependent.

10.2.2.2 File Type Attribute.

The file type attribute specifies the format in which the data in the file is represented. The two file types are:

1. DISPLAY - Displayable or printable character strings. Each data record corresponds to one print line.

2. INTERNAL - Data in INTERNAL machine format.

The file type attribute is internal to the application program. It is merely stored and passed on by the DSR as a distinction between two data types, without affecting the actual data stored.

10.2.2.3 Mode of Operation.

A file is opened for a specific mode of operation, specified in the OPEN I/O call. The four modes of operation are:

1. INPUT - The contents of the file may be read, but may not be altered.

2. OUTPUT - The file is being created. Its contents may be written but not read.

3. UPDATE - The contents of the file may both be written and read. Note that this mode of operation will generally only be supported by random access file structured devices or non-file structured devices.

4. APPEND - New data may be added at the end of the file, but the contents of the file may not be read.

Each DSR decides whether or not a specific mode for an I/O operation can be accepted by the corresponding device. For example, the TI Thermal Printer can only be opened in OUTPUT mode.

10.2.2.4 Temporary Files.

In the subsets of TI standard BASIC used for the 99/4 and 4A Home Computer, the file-life attribute is not implemented. Therefore the File Management System does not support temporary files, so all files are permanent by definition.

10.3 Implementation

As mentioned in section 10.2, the DSRs should present a uniform interface between the File Management System and the peripherals. This section will give implementational details on this interface. Some remarks are being made on a possible implementation of the file system for random access devices. However, no details are given for such an implementation.

10.3.1 Peripheral Access Block Definition.

All DSRs are accessed through a Peripheral Access Block (PAB). The definition for these PABs is the same for every peripheral. The only difference between peripherals, as seen by any application program, is that some peripherals will not support every option provided for in the PAB.

All PABs are physically located in VDP RAM. They are created before the OPEN call, and are not to be released until the I/O has been closed for that device or file.

Figure 10-1 shows the layout of a PAB. The PAB has a variable length, depending upon the length of the file descriptor. The meaning of the data within the PAB is explained below.

Byte	Bit	Meaning
0	All	I/O opcode - Contains opcode for the current I/O-call. A description of the valid opcodes will be given in section 10.3.2.
1	All	Flag/Status - File-type, mode of operation and data-type is stored in this byte. The meaning of the bits within this flagbyte is:

```
           *-------------------------------*
       Msb | 0   1   2   3   4   5   6   7 | Lsb
           *-------------------------------*
             |   |   |   |   |   |   |   |
             |   |   |   |   |   |   |   `-- Filetype
             |   |   |   |   |   `---`------ Mode of operation
             |   |   |   |   `-------------- Datatype
             |   |   |   `------------------ Recordtype
             `---`---`---------------------- Errorcode
```

Byte	Bit	Meaning
1	0-2	Errorcode - These three bits indicate, in combination with the I/O opcode, the error type that has occurred (0 = no error). See section 10.3.4 for error codes.
	3	Recordtype - Indicates type of record used. 0 = Fixed length records 1 = Variable length records
	4	Datatype - Indicates type of data stored in the file. 0 = DISPLAY type data 1 = INTERNAL type data
	5,6	Mode of operation - Indicates operation mode file has been opened for. 00 = UPDATE 01 = OUTPUT 10 = INPUT 11 = APPEND
	7	Filetype - Indicates file-type. 0 = Sequential file 1 = Relative record file
2,3	All	Data buffer address - Address of the data buffer in VDP RAM the data has to be written to or read from.
4	All	Logical record length - Indicates the logical record length for fixed length records, or the maximum length for a variable length record (see flagbyte).
5	All	Character count - Number of characters to be transferred for a WRITE opcode, or the number of bytes actually read for a READ opcode.

Byte	Bit	Meaning
6,7	All	Record number - Only required if the file opened is of the relative record type. Indicates the record number the current I/O operation is to be performed upon (this limits the range of record-numbers to 0 - 32767). The highest bit will be ignored by the DSR.
8	All	Screen offset - Offset of the screen characters in respect to their normal ASCII value. (Normally >60 while a BASIC is running, >00 otherwise.) This byte is used by cassettes, disks and the RS232.
9	All	Name length - Length of the file descriptor following the PAB.
10+	All	File descriptor - The device name and, if required, the filename and options. The length of this descriptor is given in byte 9.

```
*-----------------------------------------------------*
:  0                      :  1                        :
:       I/O OPCODE        :     FLAG / STATUS         :
:                         :                           :
:-----------------------------------------------------:
:  2,3                                                :
:   D A T A   B U F F E R   A D D R E S S             :
:                                                     :
:-----------------------------------------------------:
:  4                      :  5                        :
: LOGICAL RECORD LENGTH   :  CHARACTER COUNT          :
:                         :                           :
:-----------------------------------------------------:
:  6,7                                                :
:         R E C O R D   N U M B E R                   :
:                                                     :
:-----------------------------------------------------:
:  8                      :  9                        :
:     SCREEN OFFSET       :     NAME LENGTH           :
:                         :                           :
:-----------------------------------------------------:
: 10+                                                 :
:     FILE DESCRIPTOR                                 :
```

Figure 10-1 PAB Layout

10.3.2 I/O Opcodes.

This paragraph describes the valid opcodes that can be used in a PAB. Valid opcodes are shown in Figure 10-2.

Opcode	Meaning
00	OPEN
01	CLOSE
02	READ
03	WRITE
04	RESTORE/REWIND
05	LOAD
06	SAVE
07	DELETE
08	SCRATCH RECORD
09	STATUS

Figure 10-2 I/O Opcodes

The following describes the general actions invoked by an I/O-call with each of the I/O-opcodes. Each I/O-call returns any error-codes in the Flag/Status byte of the PAB.

10.3.2.1 OPEN.

The OPEN operation should be performed before any data-transfer operation except those performed with LOAD or SAVE. The file remains open until a CLOSE operation is performed. The mode of operation for which the file is to be opened must be indicated in byte 1 (Flag/Status) of the PAB. In case this mode is UPDATE, APPEND or INPUT, and a record length of zero is given in byte 4 (Logical Record Length), the assigned record length (which depends on the peripheral) is returned in byte 4. If a non-zero record length is given, it is used after being checked for correctness with the given peripheral. For OUTPUT, the record length can be specified, or a default can be used by specifying record length zero.

For any device, an OPEN operation must be performed before any other I/O operation. The DSR need only check the record length and I/O mode on an OPEN. Changing I/O modes after an OPEN may cause unpredictable results.

10.3.2.2 CLOSE.

The CLOSE operation closes the file. It informs the DSR that the current I/O sequence to that DSR has been completed. If the file or device was opened in OUTPUT or APPEND mode, an End Of File (EOF) record is written to the device or file before deallocating the PAB. After the CLOSE operation, the space allocated for the PAB may be used for other purposes. As long as a PAB is connected to an active device, the contents of that PAB must be preserved.

10.3.2.3 <u>READ</u>.

The READ operation reads a record from the selected device and copies the bytes into the buffer specified in bytes 2 and 3 (Data Buffer Address) of the PAB. The size of the buffer is specified in byte 4 (Logical Record Length) of the PAB. The actual number of bytes stored is specified in byte 5 (Character Count) of the PAB. If the length of the input record exceeds the buffer size, the remaining characters are discarded.

10.3.2.4 <u>WRITE</u>.

The WRITE operation writes a record from the buffer specified in bytes 2 and 3 (Data Buffer Address) of the PAB to the specified device. The number of bytes to be written is specified in byte 5 (Character Count) of the PAB.

10.3.2.5 <u>RESTORE/REWIND</u>.

The RESTORE/REWIND operation repositions the file READ/WRITE pointer either to the beginning of the file, or, in the case of a relative record file, to the record specified in bytes 6 and 7 (Record Number) of the PAB. This operation can only be used if the file was opened in INPUT or UPDATE mode. For relative record files, a RESTORE can be simulated in any I/O mode by specifying the record at which the file is to be positioned in bytes 6 and 7 (Record Number) of the PAB. The next I/O operation then automatically uses the indicated record.

10.3.2.6 <u>LOAD</u>.

The LOAD operation loads an entire memory image of a file from an external device or file into VDP RAM. All the control information the application program needs should be concatenated to the program image. Since no intermediary buffers are used, the LOAD operation requires as much buffer in VDP RAM as the file occupies on the diskette or other device. The entire memory image is dumped starting at the specified location.

The LOAD operation is a stand alone operation, i.e. the LOAD operation is used without a previous OPEN operation.

For this operation, the PAB needs to contain only the following information:

Byte 0 :	I/O opcode.
Bytes 2,3 :	Start address of the VDP RAM memory dump area.
Bytes 6,7 :	Maximum number of bytes to be loaded.
Byte 9 :	Name length.
Bytes 10+ :	File descriptor information.

10.3.2.7 SAVE.

SAVE is the complementary operation for LOAD. It writes memory images from VDP RAM to a peripheral. The SAVE operation is used without a previous OPEN operation. It copies the entire memory image from the buffer in VDP RAM to the diskette or other device. All necessary control information should be linked to the memory image, so that the information plus program image use one contiguous memory area. Again, only a small part of the PAB is used. The PAB contains:

Byte 0 :	I/O opcode.
Bytes 2,3 :	Start address of the VDP RAM memory area.
Bytes 6,7 :	Number of bytes to be saved.
Byte 9 :	Name length.
Bytes 10+ :	File descriptor information.

10.3.2.8 DELETE.

The DELETE operation deletes the specified file from the specified peripheral. This operation also performs a CLOSE.

10.3.2.9 SCRATCH RECORD.

The SCRATCH RECORD operation removes the record specified in bytes 6 and 7 (Record Number) of the PAB from the specified relative record file. This operation causes an error for peripherals opened as sequential files. No device currently supports this operation.

10.3.2.10 STATUS.

The STATUS operation is used for obtaining information about a file. This information can be examined at any time, although bits 6 and 7 only have meaning if a file has been opened.

To indicate the current status of the file, byte 8 (SCREEN OFFSET) is used. Upon the DSR-call, byte 8 should contain the usual screen characters base address. The DSR can only use this byte, and is guaranteed not to destroy any other entry in the PAB.

The meaning of the bits within byte 8 after return from the DSR is shown in the following table.

Table 10-1 Meaning of byte 8 after return from DSR

Bit	Information
0	If this bit is set, the requested file doesn't exist. If reset, the file does exist. On some devices, such as a printer, this bit is never set since any file could exist.
1	PROTECT flag. If set, the file is protected against modifications. If reset, the file is not protected.
2	Reserved for future use. Fixed to zero in the current peripherals.
3	Data type. If set, the data type is binary (INTERNAL). If reset, the data type is ASCII (DISPLAY) or file is program file.
4	Filetype. If set, the file is a program file. If reset, the file is a data file.
5	Record type. If set, the record type is VARIABLE length. If reset, the record type is FIXED length.
6	Physical end of file. If set, no more data can be written, since the physical limits of the device have been reached. Generally this means an end of medium has been detected on the device.
7	Logical end of file. If set, the file is at the end of its previously created contents. This is usually the case if the file has been opened for APPEND mode. Depending upon the mode of operation for which the file has been opened, data can still be written to the file

(APPEND, OUTPUT or UPDATE mode), however, a "read" operation will cause an ATTEMPT TO READ PAST EOF error to occur.

Bits 0 - 5 have meaning even if the file is not open. Bits 6 and 7 only have meaning for files that are currently open, otherwise a zero should be indicated in these two bits.

10.3.3 Directory Handling.

The GROM cartridge containing the DSR for a device that supports files, shall also contain a CATALOG program, which can be used to list the current contents of the medium.

10.3.4 Error Codes.

The File Management System supports a number of error codes. Errors are indicated in bits 0 thru 2 of byte 1 (Flag/Status) of the PAB. The following table shows the possible error codes and their meanings.

Table 10-2 Error Codes and Meanings

Error Code	Meaning
0	BAD DEVICE NAME The device indicated is not in the system.
1	DEVICE WRITE PROTECTED
2	BAD OPEN ATTRIBUTE One or more of the given OPEN attributes are illegal or do not match the file's actual characteristics. This could be: * File type * Record length * I/O mode * File organization
3	ILLEGAL OPERATION Either an issued I/O command was not supported, or a conflict with the OPEN mode has occurred.
4	OUT OF TABLE/BUFFER SPACE The amount of space left on the device is insufficient for the requested operation.
5	ATTEMPT TO READ PAST EOF This error may also be given for non-existing records in a relative record file.
6	DEVICE ERROR Covers all hard device errors, such as parity and bad medium errors.
7	FILE ERROR Covers all file-related errors like: program/data file mismatch, non-existing file opened for INPUT mode, etc.

An attempt is made to use these error-codes consistently for all 99/4 peripherals. If any deviation from the given codes is necessary, this should be clearly noted in the device's Software Functional Specification.

NOTE

Error code 0 usually indicates that no error has occurred. However, an error code of 0 with the COND bit (bit 2) set in the STATUS byte at address >837C indicates a bad device name.

10.4 DSR Operations

This section describes how a variety of DSRs should react on the different I/O calls. It also discusses detailed software operations descriptions such as available registers and memory.

10.4.1 DSR Actions and Reactions.

In the Professional/Home Computer File Management System, several assumptions are made about the way in which DSRs should react on conditions like errors, special I/O modes, defaults etc. This section is intended to explain the reactions of a DSR on these conditions.

10.4.1.1 Error conditions.

Non-existing DSRs. If a non-existing DSR is called by an application program, the File Management System will automatically return with the COND bit set. In this case, no DSR has actually been called, so the error-code will show no errors.

The DSR search mechanism of the File Management System takes care of searching for the requested DSR. It tries to match the file descriptor to the DSR entries in the system. The matching algorithm matches up to the end of the descriptor, or up to the first period ("."), whichever comes first. This enables the application program to add special information for the DSR in the file descriptor, like: filename, BAUD-rate, print-width, character set, etc.

DSR-detected errors. DSR-detected error (see section 4.4) should be indicated in the flag-byte of the PAB. It is the application program's responsibility to clear this flag-byte before every I/O-call, and check it after the I/O-call. This type of errors is NOT indicated with the COND bit.

The DSR may provide additional information about the error-type in the I/O opcode byte, although it is good practice not to destroy the least significant 4 bits of this byte, since they specify the I/O call.

At no time should the DSR use bits 0-4 of the flagbyte for error-indication, since these bits might contain vital system information about the file/device.

10.4.1.2 Special I/O modes.

To enable the application program to use special device-dependent functions, the File Management System DSR search algorithm only uses a well-defined part of the file descriptor for its search (see section 10.4.1.1). The remainder of the descriptor may be used to indicate special device-related functions such as BAUD-rate, print-width, etc. It is advisable for a DSR to ignore descriptor parts it doesn't recognize, so that the same application program might be used for different devices. In the latter case it would handle specific device-dependent functions only if the device used was capable of performing them, and it would ignore them if the DSR wouldn't recognize them.

An example of a special I/O mode descriptor could be:

RS232.BAUDRATE=1200.DATABITS=7.CHECKPARITY.PARITY=ODD

10.4.1.3 Default Handling.

Sometimes, especially if a file is opened for UPDATE, INPUT or APPEND, it is useful to provide a default value for the record length. In the above cases, the application program will usually use the value specified on file creation. Therefore, if the application program does not provide a value for the record length, the DSR should provide this value for it. In case the application program does provide a record length, the DSR should check this value against the value given on record creation. An error should be indicated in case of incompatibilities between stored and provided record length.

The DSR handles the default value, if no value is given, for the record length. TI BASIC supplies default values for other information. These default values are used only if no values are specified. The following shows these defaults.

Table 10-3 TI BASIC Default values

Possibilities	Default
Sequential or relative	Sequential
UPDATE, OUTPUT, INPUT or APPEND	UPDATE
DISPLAY or INTERNAL	DISPLAY
Fixed or variable length	Fixed, if relative and Variable, if sequential
Logical record length	Depends on the peripheral

10.4.2 Memory Requirements.

Because of the limited amount of register memory, 256 bytes of RAM, the register usage for DSRs has to be restricted to the following registers/memory, to avoid interference with application programs:

Registers R0 - R10 of the calling workspace.

If the DSR is called in a non-interrupt driven mode, i.e. through a standard DSR-entry, memory locations >DA through >DF are available.

A standard scratch area of 36 bytes, at locations >4A through >6D, has been assigned for DSR usage.

The base address for CPU memory in the 99/4 Home Computer is >8300. To allow for future changes in this base address, it will be derived from the given value of the workspace pointer. This means the loss of one of the workspace registers for this purpose.

10.4.3 GPL Interface to DSRs.

The GPL interpreter interfaces to DSRs through the monitor. The GPL program that wants to access a DSR, has to use the following GPL CALL sequence:

```
CALL >10
DATA 8
```

This will cause the monitor to start searching for a DSR with the same name as the string pointed to by CPU location >56. This search routine will stop comparing names at the end of the given string, or at the first imbedded period, whichever comes first.

On the DSR side, CPU location >56 is left pointing at the first character behind the DSR name, i.e. a period or an end of string. CPU location >54 contains the DSR name length (1 word). To get the start address of the PAB in VDP RAM, the following formula has to be computed:

CPU(>56) - CPU(>54) - >0A

The result will point at the I/O OPCODE entry in the PAB.

It is up to the DSR to check for any switches in the name. For this purpose the length of the PAB name string has been given in the PAB. Comparing this length against the length given in CPU location >54 will show if the user specified more than just the DSR name.

10.5 Linkage to BASIC

This section describes the way the BASIC versions of the Professional Computer and the Home Computer are linked to the File Management System.

This section also describes how to access PABs from GPL subroutines that are callable from BASIC and assume a PAB link-structure has been set up by BASIC.

10.5.1 BASIC PAB modifications.

Aside from the control information contained within the PAB, as already discussed in section 10.3, BASIC adds four (4) more bytes to the top of the PAB for specific BASIC related control information. The new PAB structure within BASIC is drawn in Figure 10-3.

0,1 LINK TO NEXT PAB IN CHAIN	
2 CHANNEL NUMBER	3 INTERNAL OFFSET
4 I/O OPCODE	5 FLAG / STATUS
6,7 D A T A B U F F E R A D D R E S S	
8 LOGICAL RECORD LENGTH	9 CHARACTER COUNT
10,11 R E C O R D N U M B E R	
12 SCREEN OFFSET	13 NAME LENGTH
14...	

Figure 10-3 Modified PAB Layout

The additional four bytes contain additional control information BASIC needs for its internal PAB linkage structure. The PABs within the BASIC control structure, form a simple linked lists, which means each PAB has a pointer to the next PAB in the system. The last PAB in the list has a zero ("0") link, indicating that it is at the end of the list.

The first 2 bytes in the BASIC PABs contain the link mentioned above, whereas the next two bytes control additional information. Byte 2 contains the actual BASIC channel or file number (1-255); byte 3 contains the offset of the current data-pointer within the data-block.

The offset indicated in byte 3 of the BASIC PAB, indicates the position of the current data pointer within the data buffer given in bytes 6 and 7. If byte 3 equals zero, the current data buffer is "blank", i.e. if we're in "read" mode, a new buffer has to be read in before any further processing; in "write" mode, the entire buffer is still available for data-storage.

If byte 3 is non-zero, it contains an "offset" within the data buffer. Added to the start address of the data buffer, it will give the actual address of the first data-byte to be read or written. This is only the case if we have pending PRINT operations (i.e the most recent PRINT ended on ";" or "," or pending INPUT operations) the most recent INPUT ended on a ",". In all other cases, byte 3 will be zero.

10.5.2 BASIC PAB Linkage.

As mentioned already, BASIC utilizes a simple linked structure for the management of its PABs. Each PAB contains a link to the next PAB in the chain. In order to access the chain, we need to have a link to the first PAB in that chain. This link is given in CPU location >3C for GPL programs, which is equal to location >833C in 9900 assembly language. A graphical representation of how three PABs could be linked in TI BASIC is:

```
   CPU RAM        <---------- VDP RAM ------------->

    003C          0FAB           0E27           0D1A

  ----------    ---------      ---------      ---------
 |  0FAB    |-->|  0E27   |--->|  0D1A   |--->|  0000   |
  ----------    |---------|    |---------|    |---------|
                | 04 | -  |    | 01 | -  |    | 2A | -  |
                |---------|    |---------|    |---------|
                |         |    |         |    |         |
                | PAB #1  |    | PAB #2  |    | PAB #3  |
                |         |    |         |    |         |
```

Figure 10-4 BASIC Link Structure

Note that all PABs are located in VDP memory, whereas the initial link to them is located in CPU memory. In addition, although the PABs will usually be allocated in lexical order, depending upon the program, they can be allocated in any arbitrary order. Therefore, the fact that in some applications they happen to be in lexical order according to channel numbers should never be used.

APPENDIX A

Software Development Methods

A.1 TI BASIC

TI BASIC, the resident language of the TI-99/4 and 4A, is a full floating-point, 13-digit expanded version of BASIC that is fully compatible with ANSI (American National Standards Institute) Minimal BASIC. Developing software in TI BASIC requires the least investment, yet still makes possible the use of the color, graphics, sound, speech and file-handling capabilities of the TI-99/4 and 4A. If an application requires greater computational power and limited graphics capability, this language is a good choice. TI BASIC features include:

* Dynamic string variables up to 255 characters in length.

* Three dimensional arrays of both numbers and character strings.

* In-line editing with programs with a re-numbering capability.

* Variable names up to 15 characters in length.

* English language error messages.

* A full standard 64 character set is provided as the default character set.

* Powerful program debugging features including break points and trace.

Programs written in TI BASIC can be diskette-and/or cassette-based.

A.2 TI Extended BASIC

With TI Extended BASIC, all the capabilities of TI BASIC are available, as well as these additional features:

* Input and output statements will clear the screen, accept only certain characters, limit the number of characters entered and format data on the screen and accessory devices more easily.

* Subprograms may be user-written and can be stored on a cassette tape or a diskette. Subprograms stored on a diskette can be added to programs as needed.

* A program can include sprites (specially defined graphics with the ability to move smoothly on the screen).

* A program can include statements which determine the action taken to handle errors that occur while the program is running (error handling).

* Seven dimensional arrays are now possible.

* RUN can be either a statement or a command and programs can be chained so that one program can load and run another one from a cassette tape or a diskette.

* More than one statement can be included on a line to speed program execution, save memory and allow logical units to be on a single line.

* Programs can be protected to prevent unauthorized copies and changes.

* With the Memory Expansion unit, the computer's memory capacity can be increased, thus allowing the ability to create more complex programs and to load and run TMS9900 Assembly subroutines.

Software developed in TI Extended BASIC can be diskette- and/or cassette-based.

A.3 Graphics Programming Language

GPL, or Graphics Programming Language, is an assembly-level language designed especially for the TI-99/4 to provide the best possible trade-off of code compaction, execution speed and ease of program development. Programming in GPL offers more capability for graphics and a higher execution speed than TI BASIC and TI Extended BASIC. The language is byte-oriented, and instructions typically have one or two operands. The addressing scheme allows easy access to the TI-99/4 and 4A memory (16K RAM) and the Command Module memory (30K ROM).

Programs written in GPL are developed ONLY as Command Modules.

A.4 UCSD PASCAL

The USCD Pascal programming language (developed originally at the University of California at San Diego) is a structured language that is a popular standard among software developers. With the Pascal software development system, consisting of the Pascal accessory (containing the interpreter and operating system), the Memory Expansion unit and software version IV.0 of the USCD Pascal editor and compiler, it is possible to:

* Compile the output so that programs run faster and require less memory than source-interpreted programs.

* Edit full screens of data, not just individual lines.

* Write, load and run Assembly programs.

* Develop Assembly language programs that can be loaded and run with TI Extended BASIC.

* Produce software that can run on many different computers without modification.

* Access the sound, speech, color and sprite capabilities of the TI-99/4 and 4A.

* Increase the execution speed of programs with Assembly language subroutines.

Programs written in USCD Pascal can be diskette-based, cassette- based or Command Modules.

A.5 TMS9900 Assembly

The TI-99/4 and 4A offers several methods for utilizing TMS9900 Assembly language in software applications. With the UCSD Pascal system, Assembly language programs can be written, loaded and run on the Home Computer. Also, the Pascal system allows development of Assembly language subroutines. With the utility program and the Memory Expansion unit, the user can load and run the subroutine from TI Extended BASIC. By using the line-by-line Assembler included on cassette with the TI Mini Memory Cartridge, the user can write, link and run Assembly language programs. Other peripherals are supported but are not necessary with the Mini Memory Assembler. The TMS 9900 Microprocessor Editor/Assembly Cartridge allows the user to program the Home Computer in TMS 990 Assembly Language. It enables direct access to all system features, including sound, speech, graphics and I/O, as well as provides the fastest spped possible from the computer's 16-bit microprocessor. The cartridge requires the Peripheral Expansion System, the Memory Expansion Card and the Disk Memory System with at least one disk drive.

Programs incorporating Assembly language have two major advantages over other programs - the ability to perform functions not available in the host language and a faster running speed.

Software applications written in Assembly language can be diskette-and/or cassette-based for use with UCSD Pascal or TI Extended BASIC. Programs written in Assembly language and designed to run with UCSD Pascal can also be developed as Command Modules.

APPENDIX B

Compatibility

In general there will be no designated compatibility between the TI-99/4 and any previous computers. There is no compatibility with products of Texas Instruments calculator line. The BASIC language is very similar to other microcomputer BASICs, however, most programs will need some changes to run on the 99/4. Any programs which use the graphics capabilities of other computers will need to be totally rewritten to run on the 99/4. The memory format of a BASIC program is unique as are most personal computers. The image which is recorded to mass-storage in a SAVE command is this memory image which limits the capability of transporting BASIC programs to other computers even if they could read our mass storage media.

Peripheral devices for the 99/4 including the Mini-floppy Disk and RS-232 Interface will not work on any other personal computer. Peripherals from other computers will not work on the 99/4 except for those with RS-232 interfaces which can be attached to our RS-232 peripheral. The media of other mass storage peipherals (audio tape or disk) will not be transportable to the 99/4.

The TI-99/2 BASIC program cassette tapes are compatible with the 99/4, but the 99/2 is not compatible with 99/4A Solid-State Software or vice-versa.