Widget Firmware Specification and Theory of Operation

Revision 2.0-0
May 8, 1984

Written by Rodger Mohme
MS-19F x4879

Some Useful Definitions:

The following is an explanation of the symbols that will be used throughout this document to describe the operation of the various firmware commands.

'<,>' : The bracket symbols mean that the information inclosed within them is manditory.

'[,]' : The square bracket symbols mean that the information inclosed within them is optional.

'|' : The vertical bar symbol is used to indicate an alternative or "OR" condition. For example, A|B can be thought of as "Either A or B".

'::=' : This symbol is used to indicate a definition or equivelence.

'{, }' : Curly brackets are used to denote commemnts.

'+' : The plus sign is used as an addition symbol or logical or'ing.

'$' : The dolar sign is used to indicate that a value is radix 16 {in other words, the number is in hexadecimal}. Values that are not preceded by '$' are assumed to be decimal.

'NULL' : This key word indicates the empty set, or in some cases the fact that the function whose value is NULL can be ignored. An example is:

Argle_Bargle ::= <NULL>

Essentially you can forget that Argle_Bargle exists for this context.

Command Types:

Widget commands are broken up into 3 categories:

1. ProFile commands

   These commands are used emulate a ProFile mass storage device and provide for downward compatibility.

2. Diagnostic commands

   These commands are used to seperate the various subfunctions of the drive and provide a means to troubleshoot a Widget without the controller of performing any retrying of it's own.

3. System commands

   These commands are used to operate a Widget at it's maximum efficiency. Blocks are transfered logically in a multiple block fashion, up to 255 blocks.

ProFile Commands:

Widget is designed to be backwards compatible with the current ProFile Driver, and to that end there exists the three ProFile System commands {Read, Write, and Write_Verify} within the firmware.

Opcode	Definition
$00	Read Logical Block
$01	Write Logical Block
$02	Write_Verify Logical Block

The three ProFile commands behave in exactly the same fashion as do the corresponding instructions on ProFile, with one small exception: the Read Logical command does not include information concerning Retry Count or Sparing Threshold {however, because of a side effect in the way that the Host/Controller interface was designed, the Host may write as many command bytes to the controller as it chooses. The Controller will only decode the first four.}. The form of each command is:

<$00|$01|$02> <3 bytes of Logical Block Address>

There are two 'special' logical address defined in the ProFile protocol, namely $FFFFFF {-1} and $FFFFFE {-2}. Logical address (-1) returns as it's value Device_ID {as explained under the section titles Diagnostic Commands} and logical address (-2) returns as it' s value Widget's spare table structure in it's raw form.

It should be noted that if *at any time* Widget can not pass it's self test that it will refuse to communicate via logical commands {both ProFile and System type commands}; Widget will respond to Diagnostic commands at all times, however.

The rest of the commands available on Widget are a complete departure from the way that ProFile was implemented. The new form of any command is:

(<Command_Byte>
<Instruction_Byte>
[Instruction_Parameter]
<CheckByte>)

Command_Byte ::= <CommandType_Nibble + CommandLength_Nibble>
CommandType_Nibble ::= <Diagnostic_Command|System_Command>
Diagnostic_Command ::= <$10>
System_Command ::= <$20>

CommandLength_Nibble ::= <Count of all the bytes in the command string ***NOT*** including the first one. For example, the command string to read Device_ID is: (<$12> <$00> <$ED>). The commandlength_nibble in this case is 2.>

System_Command ::= <Sys_Read|Sys_Write|Sys_WrVer>

```
Diagnostic_Command ::= ( <Read_ID|
                         Read_Controller_Status|
                         Read_Servo_Status|
                         Send_Servo_Command|
                         Send_Seek|
                         Send_Restore|
                         Set_Recovery|
                         Soft_Reset|
                         Send_Park|
                         Diag_Read|
                         Diag_ReadHeader|
                         Diag_Write|
                         Auto_Offset|
                         Read_SpareTable|
                         Write_SpareTable|
                         Format_Track|
                         Initialize_SpareTable|
                         Read_Abort_Stat|
                         Reset_Servo|
                         Scan> )
```

Instruction_Parameter ::= { This value is instruction dependent, and will be formally defined at the same time as the individual instructions }

CheckByte ::= { This byte is the ones-complement of the sum, in MOD-256 arithmetic, of all the bytes in the instruction string *including* the Command_Byte. }

Diagnostic_Commands:

Widget's personality, or manner in which it behaves in a specific Host environment, can be thoght of as having two distict parts: 1) that portion that is dictated by the hardware and 2) that portion that is controlled by the firmware. As trite as that last statement may seem, the fact remains that the part of Widget that is the hardware is notr easily molded to adapt to different conditions. The same is true, but not quite in the same manner, for the firmware: the code is locked in a ROM of some sort and costs a lot to change. How then can Widget's "personality" be changed {on-the-fly} to "adapt" to a new environment? The answer in thjis case was to architect the firmware in a layered fashion: build the intelligence required to operate Widget in it's normal system mode from a pool of discrete, primitive functions; these primitive functions having just one specific task that they are capable of completing. The implication of this architecture is that with very little effort these same primitive functions are available to the Host system.

Read_ID

Read_ID ::= <$00>
Instruction_Parameter ::= <NULL>

This diagnostic command requires Widget to deliver to the host some device specific information. The structural layout of the data returned is:

STRUCTURE Identity_Block

This identity block is defined by the data structures contained within it; you will note, however, that a comment is given explaining the type of structure for a given element and range of bytes - if the structure is thought of as a linear array of bytes - that include the structure. An example is NameString. It is a 13-character ascii string, and is located in bytes $0:C.

NameString ::= <10MB_Name|
20MB_Name|
40MB_Name {13 bytes/$0:C; Ascii String}>

10MB_Name ::= <'Widget-10 '>
20MB_Name ::= <'Widget-20 '>
40MB_Name ::= <'Widget-40 '>

Device_Type ::= <Device.Widget+Widget.Size+Widget.Type {3 bytes/$D:F}>

Device.Widget ::= <$0001 {2 bytes/$D:E}>
Widget.Size ::= <Size_10|Size_20|Size_40 {4 bits, byte $F/bits 7:4}>

Size_10 ::= <$00>
Size_20 ::= <$01>
Size_40 ::= <$02>

Widget.Type ::= <System|Diagnostic|AppleBus {4 bits, byte $F/bits 3:0}>

System ::= <$00 {parallel host interface}>
Diagnostic ::= <$01 {development use only}>
AppleBus ::= <$02 {serial host interface}>

Firmware_Revision ::= <{2 bytes/$10:11}>

Capacity ::= <Cap_10|Cap_20|Cap_40 {3 bytes/$12:14}>

Cap_10 ::= <$004C00>
Cap_20 ::= <$009800>
Cap_40 ::= <$013000>

Bytes_Per_Block ::= <532 {2 bytes/$15:16}> 262

Number_Of_Cylinders ::= <Cyl_10|Cyl_20|Cyl_40 {2 bytes/$17:18}>

Cyl_10 ::= <514>
Cyl_20 ::= <514>
Cyl_40 ::= <1028>

Number_Of_Heads ::= <2 {1 byte/$19}>

Number_Of_Sectors ::= <Sctr_10|Sctr_20|Sctr_40 {1 byte/$1A}>

Sctr_10 ::= <19>
Sctr_20 ::= <38>
Sctr_40 ::= <38>

Number_Of_Possible_SpareBlocks ::= <$00004C {3 bytes/$1B:1D}>

Number_Of_SpareBlocks ::= <{3 bytes/$1E:20, range 0..$4B}>

Number_Of_BadBlocks ::= <{3 bytes/$21:23, range 0..$4B}>

Read_Controller_Status

```
Read_Controller_Status ::= <$01>
```

Every time an operation completes {normally or abnormally} Widget will return Standard_Status. This allows the Host system to change it's flow of execution based on the state of the value returned in the Status. Normally, Standard_Status is all that is necessary to ensure continuous operation. In the exceptional case, or when the Host system is emulating the controler's functions, additional information concerning the state of Widget is mandatory: without it the Host simply could not make an optimum choice in deciding a course of action.

Controller_Status is then a means for the Host system to interrogate Widget further. Each Status {with the exception of Abort_Status, which is a seperate command and is discussed later in this document} belongs to a homogeneous data structure: namely a four byte quantity containing a bit map representing the various exceptional conditions thyat are available as the first four bytes read from the controller upon completion of the current command.

There are eight status' available to the Host system. The Host requests a specific status by setting the Instruction_Parameter to the value corresponding to the status needed.

```
IF (Instruction_Byte = Read_Controller_Status)
 THEN Instruction_Parameter ::= (<Standard_Status|
                                  Last_Logical_Block|
                                  Current_Seek_Address|
                                  Current_Cylinder|
                                  Internal_Status|
                                  State_Registers|
                                  Exception_Registers|
                                  Last-Seek_Address>)
```

The four byte response to each of the above status requests is of the form:

```
Status_Response ::= (<Byte0> <Byte1> <Byte2> <Byte3>)
```

SPARE TABLE CODE

ØØ ØC Ø5 11 ØA Ø3 ØF Ø8 Ø1 ØD Ø6 12 ØB Ø4 1Ø
Ø9 Ø2 ØE Ø7

```
Standard_Status ::= <$00>

  Byte0 ::= <Bit7: Other than $55 response from Host
             Bit6: Write Buffer OverFlow
             Bit5: {not used}
             Bit4: {not used}
             Bit3: Read Error
             Bit2: No Matching Header Found
             Bit1: Servo Error
             Bit0: Operation Failed >

  Byte1 ::= <Bit7: {not used}
             Bit6: Spare Table OverFlow
             Bit5: 5 or Less Spare Blocks Available
             Bit4: {not used}
             Bit3: Controller SelfTest Failure
             Bit2: Spare Table has been Updated
             Bit1: Seek Error
             Bit0: Controller Aborted Last Operation >

  Byte2 ::= <Bit7: First Status Response since Power-On
             Bit6: Logical Block Number Out of Range
             Bit5:0 : {not used}>

  Byte3 ::= <Bit7: Read Error Detected by Ecc circuitry
             Bit6: Read Error Detected by Crc circuitry
             Bit5: Header timeout
             Bit4: {not used}
             Bit3:0 : Number of unsuccessful retries {out of 10}>
```

Last_Logical_Block ::= <$01>

Byte0 ::= {not used}

Byte1 ::= <Most Significant Block Address>

Byte2 ::= <Next Most Significant Block Address>

Byte3 ::= <Least Significant Block Address>

Current_Seek_Address ::= <$02>

Byte0 ::= <Most Significant Cylinder Address>

Byte1 ::= <Least Significant Cylinder Address>

Byte2 ::= <Head Address>

Byte3 ::= <Sector Address>

0
2

3
5

Current_Cylinder ::= <$0~~6~~3>

Byte0 ::= <Most Significant Cylinder Address>

Byte1 ::= <Least Significant Cylinder Address>

Byte2 ::= <Head Address>

Byte3 ::= <Sector Address>

Internal Status ::= <$04>

Byte0 ::= <Bit7: Recovery On
Bit6: Spare Table Almost Full
Bit5: Buffer Structure is Contaminated
Bit4: Power reset has just occured
Bit3: Current Standard Status is non-zero
Bit2:1 : {not used}
Bit0: Controller LED is on>

Byte1 ::= <Bit7: On_Track
Bit6: Read Headers after data recal
Bit5: Current operation is a write operation
Bit4: Heads are parked
Bit3: Sequential look-ahead table search
Bit2: {not used}
Bit1: Seek_Complete
Bit0: Auto_Offset is ON>

Byte2 ::= {this status is valid ONLY after a ProFile or System Command}
<Bit7: Seek_Needed
Bit6: Head_Change_Needed
Bit5:2 {not used}
Bit1: Current block is a BAD block
Bit0: Current block is a SPARE block>

Byte3 ::= <SpareTable_Type|UserData_Type>
SpareTable_Type ::= <$08>
UserData_Type ::= <$02>

```
State_Registers ::= <$05>

  Byte0 ::= {not used}

  Byte1 ::= <Bit7: Ram_Failure
            Bit6: Eprom_Failure
            Bit5: Disk_Speed_Failure
            Bit4: Servo_Failure
            Bit3: Sector_Count_Failure
            Bit2: State_Machine_Failure
            Bit1: Read_Write_Failure
            Bit0: No_SpareTable_Found>

  Byte2 ::= <Bit7: Disk Read/-Write
            Bit6: SioRdy
            Bit5: Msel1
            Bit4: Msel0
            Bit3: Bsy
            Bit2: Cmd
            Bit1: EccError {active low}
            Bit0: Start {active low}>

  Byte3 ::= <Bit7: CrcError {active low}
            Bit6: Write_Not_Valid {active low}
            Bit5: ServoReady
            Bit4: ServoError
            Bit3:0 : Current state of the state-machine>
```

Exception_Registers ::= <$06>

```
Byte0 ::= <Bit7: Read error
          Bit6: Servo error while reading                                2
          Bit5: At least one successful read in last retry sequence
          Bit4: Header Timeout
          Bit3: CrcError or EccError                                     0
          Bit2:0 : {not used}>

Byte1 ::= <Bit7 ::= EccError                                             0
          Bit6 ::= CrcError
          Bit5 ::= Header Timeout
          Bit4 ::= {not used}
          Bit3:0 : {number of bad retries out of 10}>                    0

Byte2 ::= <Bit7: Write Error
          Bit6: Servo Error while writing
          Bit5: At least one sucessful write in last retry sequence      2
          Bit4: Header Timeout                                           0
          Bit3:0 : {not used}>

Byte3 ::= {number of bad retries out of 10}                              0
                                                                         0
```

EXCEPTION REGISTERS 07

BYTE 0 0
1

BYTE 1 5
7

BYTE 2 0
1

BYTE 3 1
1

Read_Servo_Status

Read_Servo_Status ::= <$02>

Instruction_Parameter ::= <0..8>

This status command is used to interrogate the Servo Processor in much the same way that Read_Controller_Status is used. In fact, the form of the result is the same four byte-mapped quantity.
This command is of the particular value to a diagnostician that is interested in 'scoping-out' the servo subsystem.
A more complete description of the servo commands can be read in the document titled "Widget Servo Functional Objective" written by Jim Reed.

Send_Servo_Command

Send_Servo_Command ::= <$03>

Instruction_Parameter ::= (<Byte0> <Byte1> <Byte2> <Byte3>)

Normally, the Host will allow the controller to manipulate the servo processor in order to perform useful work. For example, let's suppose that the Host system wishes to move drive's heads from one track to another. Under normal operating conditions the preferred way to perform this task is to use the Send_Seek command {explained later}. However, the Host has the capability to bypass the controller and direct the servo processor. Indeed, the Host can issue the servo command to position the heads so that the seek is completly transparent to the controller. The implication of this command is that the Host can gain even more control of the system if it so chooses.

A more complete description of the servo commands can be read in the document titled "Widget Servo Functional Objective" written by Jim Reed.

Byte0 ::= <S_Command + S_Direction + Hi_Magnitude>

S_Command ::= <Offset|
Diagnostic|
DataRecal|
FormatRecal|
Access|
Access_Offset|
Home>

Offset ::= <$10>
Diagnostic ::= <$20>
DataRecal ::= <$40>
FormatRecal ::= <$70>
Access ::= <$80>
Access_Offset ::= <$90>
Home ::= <$C0>

S_Direction ::= <Positive|Negative>

Positive ::= <$04 {towards inside diameter}>
Negative ::= <$00 {towards outside diameter}>

Hi_Magnitude ::= <0..3 {move heads in multiples of 256}>

Byte1 ::= <Low_Magnitude ::= 0..255>
{note: Hi_magnitude, Low_magnitude, and S_Direction establish the *relative* distance the heads must move to arrive at the target track}

Byte2 ::= <Offset_Direction + Auto_Offset_Switch + Offset_Magnitude>

Offset_Direction ::= <Positive|Negative>

Positive ::= <$80 {towards outside diameter}>
Negative ::= <$00 {towards inside diameter}>

```
        Auto_Offset_Switch ::= <ON|OFF>

          ON ::= <$40 {assert fine positioning}>
          OFF ::= <$00>

        Offset_Magnitude ::= <0..32>

Byte3 ::= <Baud_Rate + Power_On_Reset>

        Baud_Rate ::= <19.5k_Baud|57.6k_Baud>

          19.5k_Baud ::= <$00>
          57.6k_Baud ::= <$80>

        Power_On_Reset ::= <$40>
```

Send_Seek

Send_Seek ::= <$04>

Instruction_Parameter ::= (<HiCyl> <LoCyl> <Head> <Sector>)

Widget's Send_Seek command allows the Host system to place the heads over any track on the disk. The value of the seek address is sent as the Instruction_Parameter, and each parameter is a byte in length. For example, for the Host to seek to (Cylinder 1, Head 0, Sector 18) a seek command would be issued with the following Instruction_Parameter: ($0000, $00, $12).

Send_Restore

Send_Restore ::= <$05>

Instruction_Parameter ::= <DataRecal|FormatRecal>
DataRecal ::= <$40>
FormatRecal ::= <$70>

The Send_Restore command is used by the Host to initialize the servo processor and to put the heads in a known location. This command is the same as performing a Data/Format Recal except that the controller updates it's internal state to account for the new servo position.

Set_Recovery

Set_Recovery ::= <$06>

Instruction_Parameter ::= <ON|OFF>
ON ::= <$01>
OFF ::= <$00>

The exception handling characteristics of Widget approximate a binary set: either Widget handles everything, or the Host system does. The command 'Set_Recovery' is the Host's link with this protocol in that it is through this instruction that the Host can gain control of the media. When Widget comes up after being reset, it assumes control and sets *Recovery* to be ON. The Host system must overtly change this state if it wishes to emulate a different exception handling criteria. Once Recovery is OFF, the controller will always fail in an operation if an exception occurs: the Host *must* assume responsibility for ALL error handling.

Soft_Reset

Soft_Reset ::= <$07>

Instruction_Parameter ::= <NULL>

This command instructs the Widget firmware to restart its flow of execution at its initialization point. The results should be the same as a power reset.

Send_Park

Send_Park ::= <$08>

Instruction_Paramter ::= <NULL>

When the Host issues a Send_Park command to the controller the results are that the heads are moved off the data surface and held very near the inside diameter crash stop. The difference between this command and the Send_Servo_Command: Home, is that Home is performed 'open-loop' with the crash stop as its reference point, while Send_Park is an access command to a specific track. The net result is a fairly hefty savings of time.

Diag_Read

Diag_Read ::= <$09>

Instruction_Parameter ::= <NULL>

The Diag_Read command is used to read the block on the disk pointed to by the last seek address. The form of the returned data is exactly the same as that of ProFile_Read or Sys_Read in that 4 bytes of Standard_Status precede the block of data.

Diag_ReadHeader

```
Diag_ReadHeader ::= <$0A>

  Instruction_Parameter ::= <Sector>
```

When the heads are positioned over an unknown location, or when it is suspected that a block's header is shot, it is time to use the Diag_ReadHeader command. This instruction allows the host to 'suck-up' both whatever information is residing in the block's header field as well as the data from the block. The form of the result is:

```
  Result ::= (<Header {bytes/$00:05}>
             <Gap {bytes/$06:0C}>
             <Data {bytes/$00:21F}>)

    Header ::= (<HiCyl> <LowCyl> <HdSct> <-HiCyl> <-LowCyl> <-HdSct>)

      HiCyl ::= <Most significant byte of cylinder address>
      LowCyl ::= <Least significant byte of cylinder address>
      HdSct ::= <Bit7:6 : Head address
                 Bit5:0 : Sector address>

      -HiCyl ::= <ones-complement of HiCyl>
      -LowCyl ::= <ones-complement of LowCyl>
      -HdSct ::= <ones-complement of HdSct>

    Gap ::= <$00>
```

Diag_Write

Diag_Write ::= <$0B>

Instruction_Parameter ::= <NULL>

This instruction allows the Host to write a block of data to the location on the disk pointed to by the last seek address. Diag_Write is valid for all states that the controller may wid up in, but is recommended that a Send_Seek command precede the write command to ensure that the correct block will be written.

Auto_Offset

Auto_Offset ::= <$0C>

Instruction_Parameter ::= <NULL>

This command is used by the Host to fine-position the heads after they are on-track. The auto_offset function can also be implemented by using the Send_Servo_Command instruction; the difference is that the controller will update some internal information {remember, servo commands are transparent} as well as select the correct head to offset off of {the Widget system uses head 1 only for fine positioning}.

Read_SpareTable

Read_SpareTable ::= <$00>

Instruction_Parameter ::= <NULL>

Reading {and writing} the Widget's sparetable is an absolute must for diagnostic purposes, and if the Host wishes to emulate the controller. The result of this instruction is identical to performing a ProFile_Read from block -1 {$FFFFFE} and has the form:

Result ::= (<Fence {bytes/$00:03}>
<RunNumber {bytes/$04:07}>
<Format_Offset {byte/$08}>
<Format_InterLeave {byte/$09}>
<HeadPtr_Array {bytes/$0A:89}>
<SpareCount {byte/$8A}>
<BadBlockCount {byte/$8B}>
<BitMap {bytes/$8C:95}>
<Heap {bytes/$96:1C5}>
<InterLeave_Map {bytes/$1C6:1D8}>
<CheckSum {bytes/$1D9:1DA}>
<Fence {bytes/$1DB:1DE}>
<Zone_Table {bytes/$1DF:1FF}>
<Fence {bytes/$200:203}>)

Fence ::= (<$F0> <$78> <$3C> <$1E>)

RunNumber ::= <32-bit integer>
This integer is incremented once each time the spare table is written to to the disk. Because two copies are kept on the the disk, the RunNumber is used to indicate which is the more recent of the two, should both copies not be updated.

Format_Offset ::= <0..NumberOfSectors>
Format_Offset is the number of physical sectors there are from index mark until logical sector 0.

Format_InterLeave ::= <0..6>
This number is the interleave factor for this disk and is used in calculating where each of the logical sectors are relative to actual sector locations.

HeadPtr_Array ::= <ARRAY[0..127] of HeadPtr

HeadPtr ::= <Nil+Ptr>
Nil ::= <$80 {if Nil the end-of-chain}>
Ptr ::= <$00..$7F {address of next element}>
A Ptr is a 7-bit structure that 'points' to a specific location within the Heap. To arrive at the actual index value within the Heap, the Ptr must first be multiplied by 4 {the length of each element}.

When a disk is formatted and being written to for the first time, each logical block is assigned the first available physical block on the disk. Therefore you would expect that LogicalBlock(0) would occupy PhysicalBlock(0), L(1) --> P(1), etc. There are instances, however, when a block of data must be relocated to anaother space on the disk that does not follow the original progression (for example, the original space was defective). In order to 'find' these relocated blocks in the future a record must be kept as to where all these relocated blocks have been put. This record takes the form of 128 linked lists having the form:

HeadPtr[n] --> LinkedList[n], where n ::= [0..127]

The algorithm for deciding whether or not a logical block has been relocated is to extract bits 10:16 from the LogicalBlockNumber and use it as an index into the HeadPtrArray:

```
IF (HeadPtr[LogicalBlockNumber/bits 10:16].Nil)
  THEN LogicalBlock has not been relocated
  ELSE use HeadPtr[].Ptr to begin searching the chain for a matching
       element {refer to the structure of ListElement for more detail}
   IF no matching ListElement
    THEN LogicalBlock has not been relocated
    ELSE the element position in the Heap corresponds to the new physical
         block location
```

SpareCount ::= <$00..$4B>

BadBlockCount ::= <$00..$4B>

BitMap ::= <ARRAY[$00..$4B] of Bits>

The bit map is used to keep a record of which spare blocks are occupied.

Heap ::= <ARRAY[$00..$4B] of ListElement>

ListElement ::= (<Nil+Used+Useable+Spr_Type+Data_Type>
<Token>
<Ptr>)

Used ::= <$40>
Useable ::= <$20>
Spr_Type ::= <Spare|BadBlock>
Spare ::= <$10>
BadBlock ::= <$00>
Data_Type ::= <Data|SpareTable>
Data ::= <$02>
SpareTable ::= <$08>

Token ::= <Bits 0:9 of LogicalBlock>

InterLeave_Map ::= <ARRAY[0..15] of [0..NumberOfSectors]>

The InterLeave_Map is used to logical re-interleave the drive so that Widget can be run optimally on any system without having different manufacturing or formatting processes.

Check_Sum ::= <sum of all bytes in the spare table from the first fence to beginning of this structure, in MOD-65536 arithmetic>

Zone_Table ::= <ARRAY[0..NumberOfZones] of Zone_Element>

Zone_Element ::= <Offset_Direction+Offset_Magnitude>

Write_SpareTable

Write_SpareTable ::= <$0E>

Instruction_Parameter ::= (<$F0> <$78> <$3C> <$1E>)

This command allows the Host to 'force' a new spare table on the controller, and is executed just like any of the other write commands (data, in this case, MUST conform to the structure presented in Read_SpareTable}. The data sent to the controller is written to the two spare table locations on the disk.

Format_Track

Format_Track ::= <$0F>

Instruction_Parameter ::= (<Format_Offset>
<Format_InterLeave>
<PassWord>)

Format_Offset ::= <0..NumberOfSectors>
This parameter dictates which sector {beginning with sector 0 - the first physical sector after index mark} will be logical sector 0 for that track.

Format_InterLeave ::= <0..6 {interleave factor}>

PassWord ::= (<$F0> <$78> <$3C> <$1E>)

The format command is used to:

1. Operate on the track that is currently beneath the heads - this implies that the Host had best perform a Send_Seek and Auto_Offset command prior top formatting a track.

2. AC erase the entire track - this implies that all data stored on this track will be destroyed.

3. New headers will be layed down in every sector of the track.

0001F0783C1E

Initialize_SpareTable

Initialize_SpareTable ::= <$10>

Instruction_Parameter ::= (<Format_Offset>
<Format_InterLeave>
<PassWord>)

Format_Offset ::= <0..NumberOfSectors>
This parameter dictates which sector {beginning with sector 0 - the first physical sector after index mark} will be logical sector 0 for that track.

Format_InterLeave ::= <0..6 {interleave factor}>

PassWord ::= (<$F0> <$78> <$3C> <$1E>)

This command instructs the controller to 'wipe the slate clean' as far as the SpareTable is concerned. The initialized table is updated on the disk.

Read_Abort_Status

Read_Abort_Status ::= <$11>

Instruction_Parameter ::= <NULL>

Read_Abort_Status will return vaild data only AFTER the controller has aborted (identified by Standard_Status.Byte1.Bit0}. The form of the result is a 16 byte string, and its contents are the contents of the controller's registers at the time of the abort - with the exception of bytes $0E:0F, which constitute the reurn address of the procedure that called the Abort routine.

Reset_Servo

Reset_Servo ::= <$12>

Instruction_Parameter ::= <NULL>

Reset_Servo allows the Host to initialize the servo processor without having to power the device down. The controller will automatically reset the Servo, set the baud rate at 57.6K, and check for valid initial conditions.

Scan

Scan ::= <$13>

Instruction_Parameter ::= <NULL>

The scan command causes the Widget to read all blocks that are within the range of blocks set aside for user data blocks (all logical blocks). If any of these blocks are bad they will be either relocated or marked as bad andrelocated on the next write. The SpareTable can be examined before and after a Scan command to find the locations of all bad blocks.

System Commands:

System commands have been implemented for essentially two reasons:

1. It was important for Widget to add one more check on the CMD/BSY handshake: namely the addition of a checkbyte following the command string.

2. In order to increase the performance of the system without modifying the hardware it was critical to introduce another level of parallelism into the Host/Controller interface. Most of the reads for a specific block on the disk are followed by a read for the next logically sequential block. Therefore the command decoding and checkbyte comparison for all but the first block has been suppressed into a multiblock-type command. The implementation for this added parallelism is to send an extra parameter with the (first) LogicalBlock indicating the number of blocks to be read sequentially.

Sys_Read

Instruction_Parameter ::= (<BlockCount> <LogicalBlock>)

BlockCount ::= <$01..$FF>

This parameter is the number of blocks to be read that follow sequentially from LogicalBlock. It is assumed that one block (LogicalBlock) will be read.

LogicalBlock ::= <L_10MB|L_20MB|L_40MB>

L_10MB ::= <$000000..004BFF>
L_20MB ::= <$000000..0097FF>
L_40MB ::= <$000000..012FFF>

Sys_Write

Instruction_Parameter ::= (<BlockCount> <LogicalBlock>)

BlockCount ::= <$01..$FF>

This parameter is the number of blocks to be read that follow sequentially from LogicalBlock. It is assumed that one block (LogicalBlock) will be read.

LogicalBlock ::= <L_10MB|L_20MB|L_40MB>

L_10MB ::= <$000000..004BFF>

L_20MB ::= <$000000..0097FF>

L_40MB ::= <$000000..012FFF>

Sys_Write_Verify

Instruction_Parameter ::= (<LogicalBlock>)

BlockCount ::= <$01..$FF>
This parameter is the number of blocks to be read that follow sequentially from LogicalBlock. It is assumed that one block (LogicalBlock) will be read.

LogicalBlock ::= <L_10MB|L_20MB|L_40MB>
L_10MB ::= <$000000..004BFF>
L_20MB ::= <$000000..0097FF>
L_40MB ::= <$000000..012FFF>

Command Summary

ProFile_Commands:

```
ProFile_Read ::= (<$00> <3 bytes LogicalBlock>)
ProFile_Write ::= (<$01> <3 bytes LogicalBlock>)
ProFile_WrVer ::= (<$02> <3 bytes LogicalBlock>)
```

Diagnostic_Commands:

```
Read_Id ::= (<$12> <$00> <$ED>)
Read_Controller ::= (<$13> <$01> <StatusRequest> <CheckByte>)
Read_Servo_Status ::= (<$13> <$02> <StatusRequest> <CheckByte>)
Send_Servo_Command ::= (<$16> <$03> <CommandRequest> <CheckByte>)
Send_Seek ::= (<$16> <$04> <SeekAddress> <CheckByte>)
Send_Restore ::= (<$13> <$05> <On/Off> <CheckByte>)
Set_Recovery ::= (<$13> <$06> <RecalType> <CheckByte>)
Soft_Reset ::= (<$12> <$07> <$E6>)
Send_Park ::= (<$12> <$08> <$E5>)
Diag_Read ::= (<$12> <$09> <$E4>)
Diag_ReadHeader ::= (<$13> <$0A> <Sector> <CheckByte>)
Diag_Write ::= (<$12> <$0B> <$E2>)
Auto_Offset ::= (<$12> <$0C> <$E1>)
Read_SpareTable ::= (<$12> <$0D> <$E0>)
Write_SpareTable ::= (<$16> <$0E> <PassWord> <CheckByte>)
Format_Track ::= (<$18> <Offset> <InterLeave> <PassWord> <CheckByte>)
Init_SpareTable ::= (<$18> <Offset> <InterLeave> <PassWord> <CheckByte>)
Read_Abort_Status ::= (<$12> <$11> <$DC>)
Reset_Servo ::= (<$12> <$12> <$DB>)
Scan ::= (<$12> <$13> <$DA>)
```

System_Commands:

```
Sys_Read ::= (<$26> <$00> <BlockCount> <LogicalBlock> <CheckByte>)
Sys_Write ::= (<$26> <$01> <BlockCount> <LogicalBlock> <CheckByte>)
Sys_WrVer ::= (<$25> <$02> <LogicalBlock> <CheckByte>)

PassWord ::= (<$F0> <$78> <$3C> <$1E>)
```

Format & S.T.

Abort_Status_Variables

There are occasions when the Widget Controller will detect that something is radically wrong with the Widget SubSystem, i.e., the ram on board the controllergoes on vacation, or the positioning system gives up the ghost, etc. In one of these cases the controller will abort its current instruction and return control to the Host, hopefully with enough information that the Host can make an intelligent decision concerning the state of Widget.

The Host can read some information concerning the abort that the controller took by requesting Read_Abort_Status. This command returns a result that is 20 bytes long: 4 bytes of standard status and 16 bytes of abort status. The contents of the abort status are dependent upon the actual abort taken, and is determined by examining the contents of bytes 15 and 16: the pointers to area of the firmware where the abort occured.

In the following table, the contents of bytes 15 and 16 are indicated {as a hexadecimal 16-bit integer, just as you would read them from the buffer} with a brief description of the reason why the abort was taken as well as any comments concerning other bytes of immediate interest included in the Abort_Status structure.

$02EA: Illegal interface response, or Host Nak
 Byte/$09: Response byte that caused abort
$03B8: Illegal Ram_Bank select
 Byte/$00: Bank number
$048A: Format Error: illegal state-machine state
 Byte/$0A: state of state-machine at time of abort
$04CE: Illegal Bank Switch (cont)
 Byte/$00: Bank number
$0516: Illegal interrupt or DeadMan_Timeout
 Bytes/$0A:0B: Address of routine at time of timeout
$1114: Format Error: Error while writing sector
 Byte/$09: Error status from FormatBlock
$1204: Command Checkbyte Error
$1216: ProFile or System command attempted while SelfTest Error
$122A: Illegal interface instruction
$1329: Unrecoverable Servo Error while reading
$1408: Sparing attempted on non-existent spare block
$1542: Sparing attempted while sparetable full
$15B8: Deletion attempted of non-existent bad block
$16E0: Illegal exception instruction
$18E8: Write buffer overflow
$192C: Unrecoverable servo error while writing
$1B0A: Servo status request sent as Servo command
$1B5F: Restore Error: Non-Recal parameter
 Byte/$00: Illegal parameter sent
$1BC3: Illegal password sent to Write_SpareTable_Command
$1C00: Illegal password sent to Format command
$1C0F: Illegal format parameters
 Bytes/$09:0A: illegal parameters
$1C63: Illegal password sent to Init_SpareTable_Command
$1CF8: Zero block count sent to System_Command

$1E49: Write Error: Illegal state-machine state
Byte/$0A: State-machine state at time of abort
$1F3C: Read Error: illegal state-machine state
Byte/$0A: State-machine state at time of abort
$2026: ReadHeader Error: illegal state-machine state
Byte/$0A: State-machine state at time of abort
$21E7: Request for illegal logical block
Bytes/$00:02: logical block number
$226F: External Stack overflow
Bytes/$04:07: stack history
$236D: Search for SpareTable failed
$2493: No sparetable structure found in sparetable
$24B3: Update of sparetable failed
$2525: Illegal sparecount instruction
Bytes/$09: value of illegal instruction
$264A: Unrecoverable servo error while seeking
$2858: Unable to transmit command to servo
$2877: Unable to receive status from servo
$2940: Unable to find any headers after DataRecal
$29C0: Servo error after servo reset
Byte/$0A: value of controller status port
$29F5: Servo communication error after servo reset
$2CD2: Scan attempted without sparetable

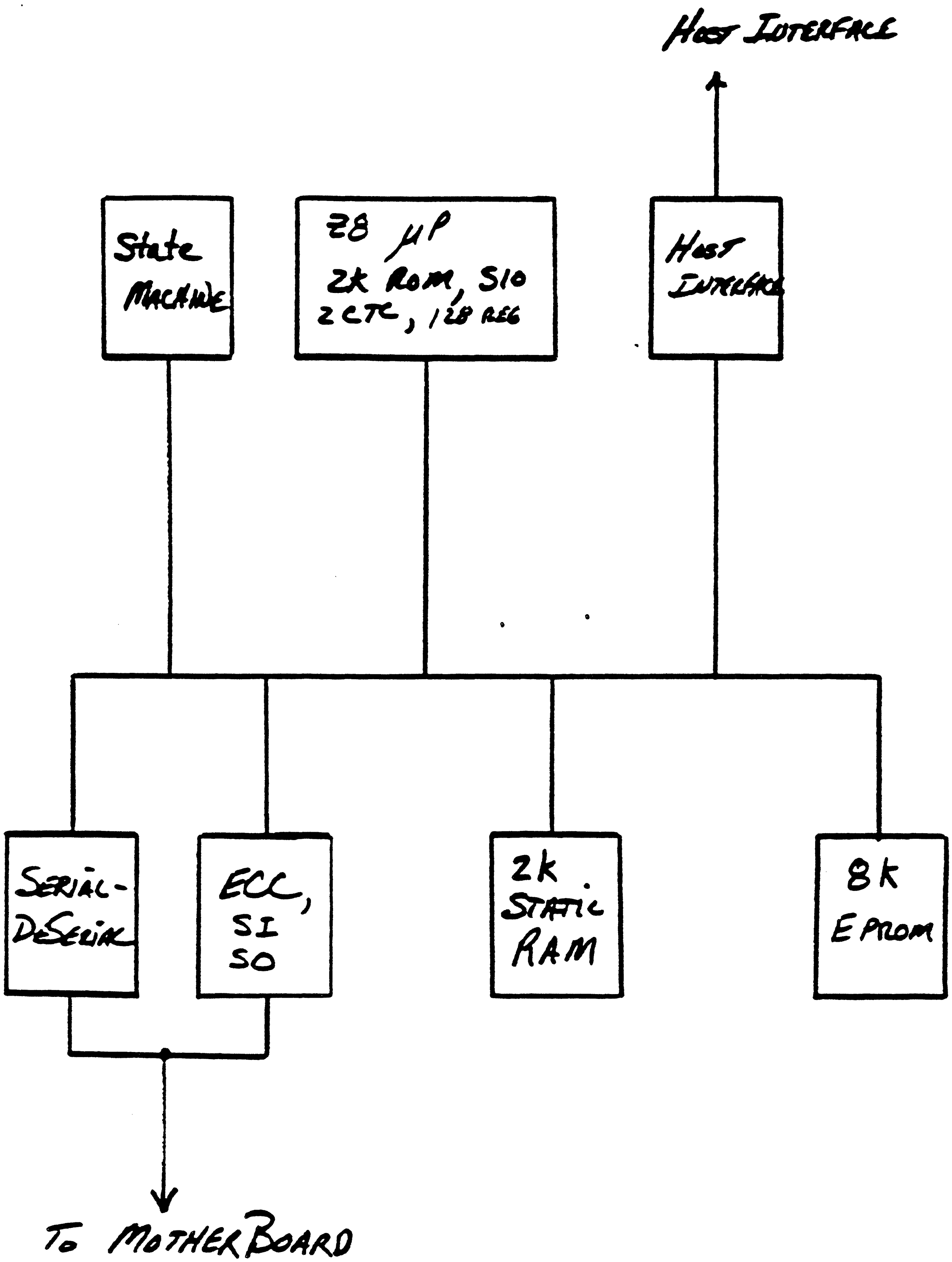
Z8

1. INTELLIGENT CONTROLLER
   a) μ COMPUTER : RAM, ROM, SIO, CTC
   b) 4 MHZ {7.3 ÷ 2}

2. RECOVERY
   a) DEFECTS → SPARING
   b) NOISE
   c) SERVO ERRORS
   d) DATA CORRECTION

BLANCHARD.

WIDGET CONTROLLER BLOCK DIAGRAM



STATE MACHINE

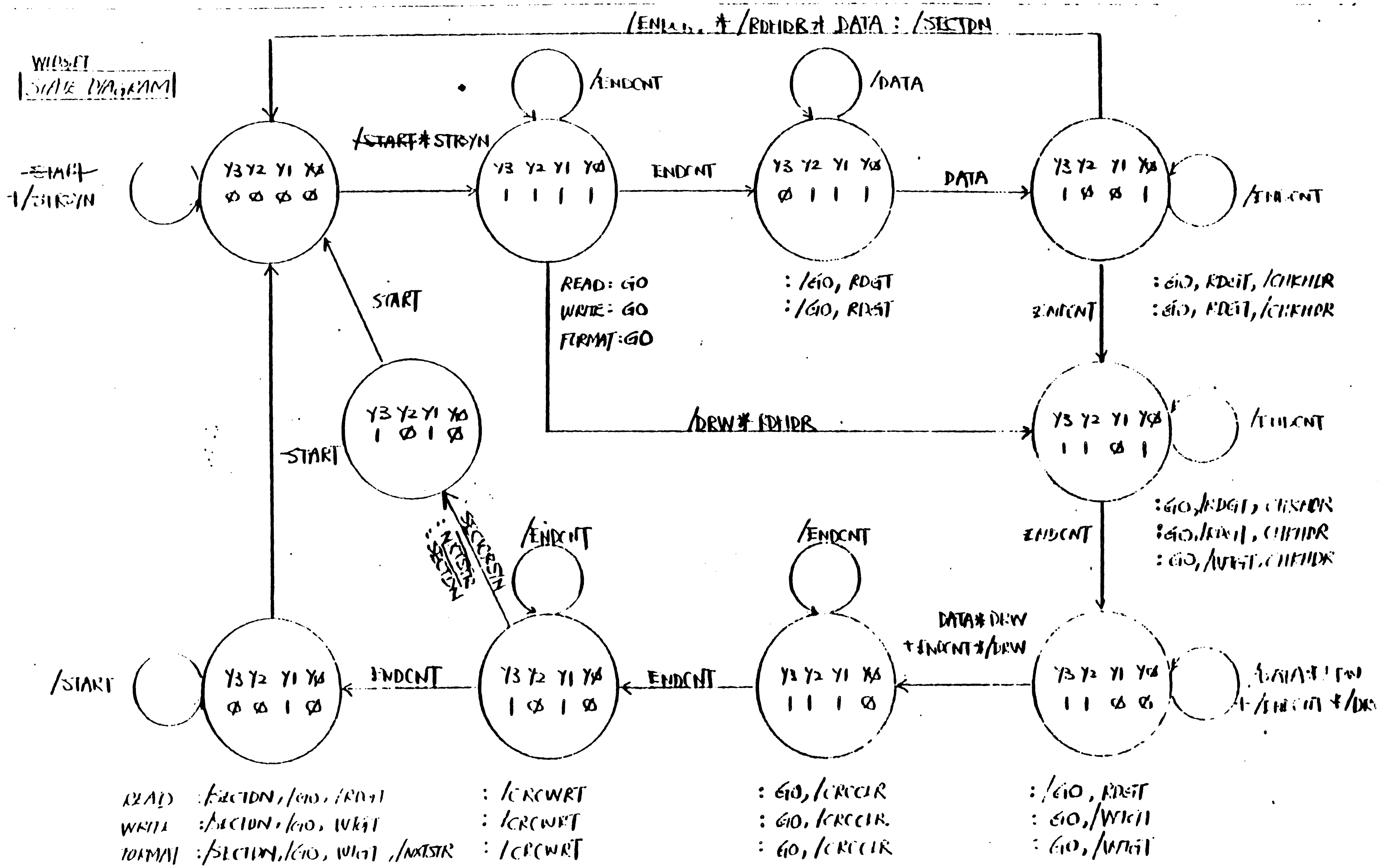
1. SYNCHRONIZATION TO DISK
2. PERFORMS READ, WRITE, FORMAT, READ HEADER
3. CRC/ECC GENERATION
   a) ERROR DETECTION
4. LOADS/STORES WRITE/READ DATA TO/FROM DISK

(5) POWER OK
   a) DETECTS WHEN +5V IS WITHIN RANGE

Z8

1. INTELLIGENT CONTROLLER
   a) μ COMPUTER : RAM, ROM, SIO, CTC
   b) 4 MHZ {7.3 ÷ 2}

2. RECOVERY
   a) DEFECTS → SPARING
   b) NOISE
   c) SERVO ERRORS
   d) DATA CORRECTION



WDSET
STATE DIAGRAM
/START * STRYN
/ENDCNT
ENDCNT
/DATA
DATA
Y3 Y2 Y1 YØ
Ø Ø Ø Ø
1 1 1 1
Ø 1 1 1
1 Ø Ø 1
1 1 Ø 1
1 Ø 1 Ø
Ø Ø 1 Ø
1 Ø 1 Ø
1 1 1 Ø
1 1 Ø Ø
READ: GO
WRITE: GO
FORMAT: GO
: /GO, RDGT
/DRW * RDHDR
START
/START
DATA * DRW
ENDCNT
: /CRCWRT
: GO, /CRCCLR
: /GO, RDGT
: GO, /WRGT
/START

Z8 OPERATION: READ / READ (NO HEADER)

```
BEGIN
   MSEL1 := FALSE; MSEL0 := TRUE  { MEM <-> Z8 }
   LOAD BUFFER WITH HEADER

         <#0B> := Hi-TRACK BYTE
         <#0C> := Lo-TRACK BYTE
         <#0D> :=
                  <Hi-NIBBLE> := HEAD SELECT
                  <Lo-NIBBLE> := SECTOR NUMBER
         <#0E> := INVERT(<#0B>)
         <#0F> := INVERT(<#0C>)
         <#10> := INVERT(<#0D>)
         <#11> := #00

   SET-UP STATE MACHINE
         MSEL1 := TRUE; MSEL0 := FALSE  { MEM <-> DISK }
         DM -> OUTPUT PORT := 0
         DRWL := FALSE  {DISK READ} ; FMENL := FALSE  {NO FORMAT}
         IF NORMAL READ OPERATION
            THEN RDHDRH := FALSE
            ELSE RDHDRH := TRUE  {DON'T CARE ABOUT HEADER}
         POLL FOR SECTOR MARK  { PORT 3, BIT 1 }
         POLL FOR NOT(SECTOR MARK)
         STARTL := TRUE  { TURN STATE MACHINE ON }

   WAIT FOR SECTOR DONE OR TIMEOUT
         IF TIMEOUT THEN EXCEPTION
         IF SECTOR DONE
            THEN
               READ STATE MACHINE STATUS
                  IF STATE 0 THEN HEADER MISMATCH/GAP NOT ZERO
                  IF STATE 2
                     THEN
                        DISK DATA AT RAM ADR (#19 - #22C)
                        CRC AT RAM ADR (#22D - #22E)
                        ECC AT RAM ADR (#22F - #234)
                        IF CRC ERROR THEN EXCEPTION
                     ELSE
                        UNKNOWN STATE EXCEPTION
         STARTL := FALSE  { RESET STATE MACHINE
END
```

NOTE: IF THIS WAS A READ HEADER OPERATION THEN THE BYTES IN RAM ADDR <#0E - #1D WERE REPLACED BY THE BYTES IN THE HEADER SPACE ON THE DISK. ~~THE HEADER [illegible]~~

FIRMWARE

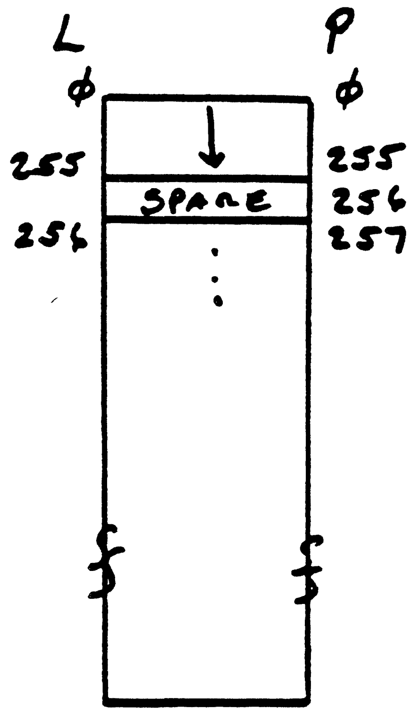
1. HOST INTERFACE PROTOCOL
    a) PROFILE, DIAGNOSTIC, MULTIBLOCK
2. CONTROLS STATE MACHINE, SERVO
    a) BASIC DISK FUNCTIONS
    b) POSITIONING
3. RECOVERY !!
4. PERFORMANCE

INITIALIZATION

1. BOOT STRAP A FEW Z8 REGISTERS
2. TEST ALL Z8 REGISTERS
3. STACK, CALL, RETURN TEST
4. INITIALIZE I/O; GLOBAL VARS
5. RAM TEST
6. EPROM TEST
7. MOTOR SPEED TEST {RELEASE BRAKE}
8. SECTOR COUNT
9. SERVO TEST
10. READ/WRITE TEST
11. FIND SPARE TABLE
12. SCAN

SPARING

Disk Blocks



1. 10 MB → 1 SPARE/256 Blocks
   20 MB → 1 SPARE/512 Blocks
   40 MB → 1 SPARE/1024 Blocks

2. A Block is SPARED iff:
   a) VALID DATA is AVAILABLE
   b) THE Block is A <u>HARD</u> DEFECT

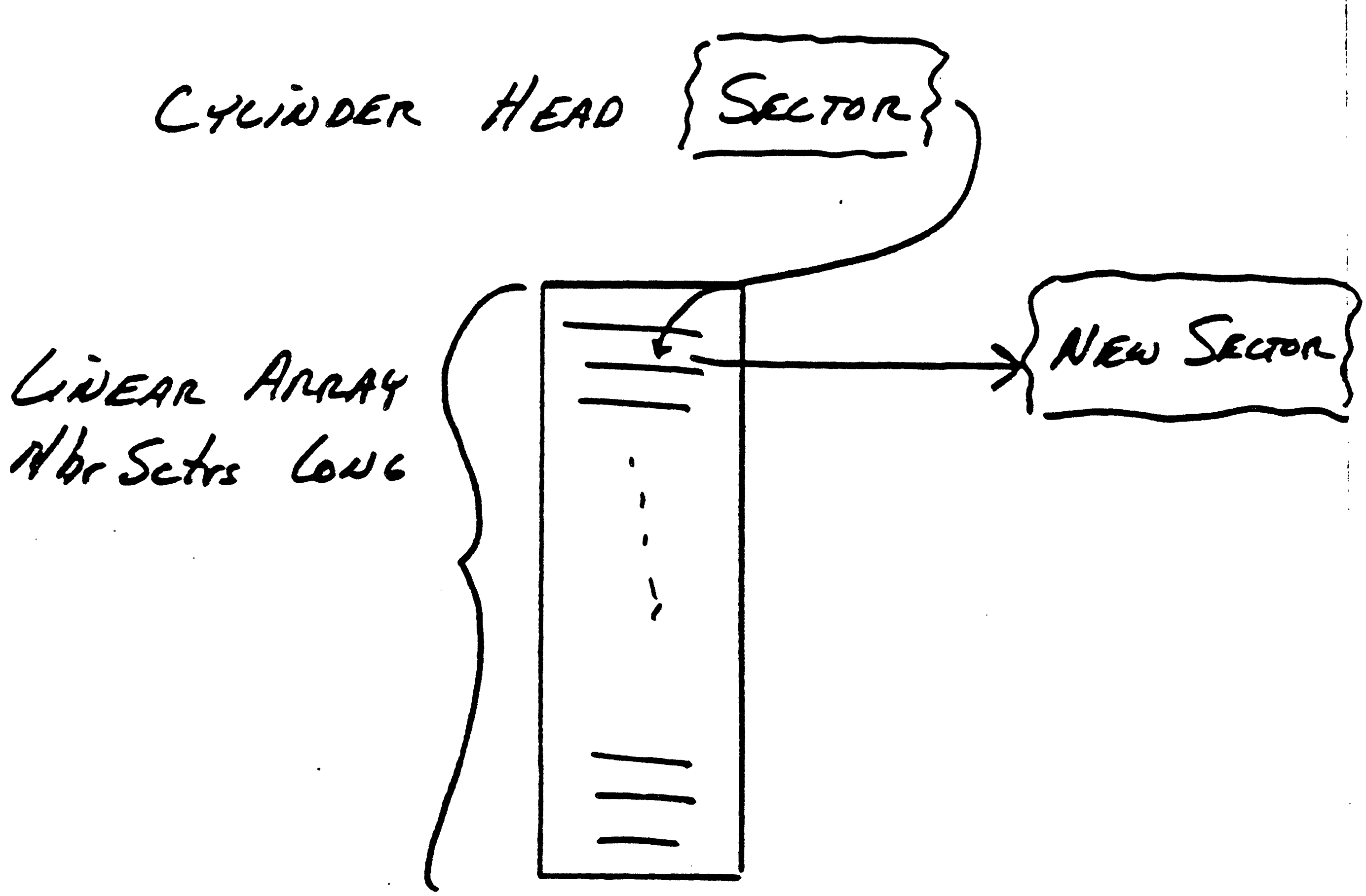
3. 76 TOTAL Blocks AVAILABLE FOR SPARING
   a) SPARE TABLE is LOCATED ON 2
   b) 74 LEFT FOR USER DATA

INTERLEAVING

1. ALL WIDGETS FORMATTED 2 : 1

2. CAPABILITY EXIST TO LOGICALLY INTERLEAVE 1:1 → Nbr Sctrs : 1

3. OFFSET SECTOR Ø
   a) UP TO 16 SECTORS
   b) HEAD Ø, HEAD 1 INDEPENDENT



BLANCHARD.

WIDGET SERVO FUNCTIONAL OBJECTIVE

I. BASIC SERVO FUNCTIONS

Widget servo control functions are handled by a Z8 microprocessor. The Z8 handles all I/O operations, timing operations and communication with a host controller. Control functions to the Z8 Servo Controller are made through the serial I/O.

The following commands for the Widget servo are:

A. HOME - not detented, heads off data zones located at the inner stop.

B. RECAL - detented at one of two positions.

   1. FORMAT RECAL: 32, -0, +3 tracks from HOME. Used only during data formatting.

   2. RECAL: 72, -0, +3 tracks from HOME. Used to initialize home position after on or following an access error or any other error.

C. SEEK - coarse track positioning of data head to any desired track location.

D. TRACK FOLLOWING - heads are detented on a specific track location and the device is ready for another command.

E. OFFSET - controlled microstepping of fine position system during TRACK FOLLOWING (two modes).

   1. COMMAND OFFSET - direction and amount of offset is specified to the servo.

   2. AUTO OFFSET - command allows the servo to automatically move off track by the amount indicated by the embedded servo signal on the data surface (disk).

F. STATUS - command can read servo status.

G. DIAGNOSTIC - not implemented.

See Table 1 for the actual command description. With the present command structure a SEEK COMMAND can be augmented with an OFFSET COMMAND. Upon completion of a seek, the offset command bit is tested to determine if an offset will occur following a seek (either auto or command offset).

When a SERVO ERROR occurs the Z8 SERVO will attempt to do a short RECAL (ERROR RECAL). Two attempts are made by the system to do the ERROR RECAL function. If either of the two RECAL operations terminate successfully the protocol status will be SERVO READY, SIO READY and SERVO ERROR. Should the ERROR RECAL fail then the system will complete the error recovery by a HOME function.

The two OFFSET commands will be described. First COMMAND OFFSET is a predetermined amount of microstepping of the fine position servo. Included in the OFFSET BYTE (STATREG), bit B6=0 is a COMMAND OFFSET. Bit B7=1 is a forward offset step (toward the spindle); B7=0 is a reverse step. If bit B6=1, the OFFSET command is AUTO OFFSET.

AUTO OFFSET command normally occurs during a write operation. When the HDA was initially formated at the factory, special encoded servo data was written on each track "near" the index zone. The reason for this follows:

Normal coarse and fine position information for the position servos is derived from an optical signal relative to the actual data head-track location. Over a period of time, the relative position (optical signal) will be misaligned to the absolute head-track position by some unknown amount (less than 100 uIn). This small change is important for reliability during the write operation. Write/Read reliability can be degraded due to this misalignment. The special disk encoded servo signal is available to the fine position servo. It will correct the difference between the relative position signal of the optics and the absolute head to track position under the data head only at index time. The correction signal can be held indefinitely or updated (if desired at each index time) until a new OFFSET command or move command (SEEK or RECAL) occurs.

II. COMMUNICATION FUNCTIONS

The servo functions described in the previous section only occur when the servo Z8 microprocessor is in the communication state. Communication states occur immediately after a system reset, upon completing head setting after a recal, seek, offset, read servo status or set servo diagnostic command. A special communication state exists after a servo error has occurred. If + SIO READY is not active, no communication can exist between the external controller and the servo Z8 processor.

Servo commands are serial bits grouped as five separate bytes total. Refer to Table 1 parts I through V for the total communication string. The first byte is the command byte (i.e. seek, read status, recal, etc.). The second byte is the low order difference for a seek (i.e. Byte 2 = $0A is a ten track seek). The third byte is the offset byte (AUTO or COMMAND OFFSET and the magnitude/direction for command offset). The fourth byte is the status and diagnostic byte (use for reading internal servo status or setting diagnostic commands). Byte five is the check sum byte used to check verify that the first four bytes were correctly transmitted (communication error checking).

Part of the communication function requires a specific protocol between the servo Z8 processor and the external controller.

Servo control and communication are described in CHART I. This chart illustrates the basic sequencing and control operations. Chart I does not illustrate the servo error handling or command/protocol handling functions. Error handling is described in Section IV and illustrated by CHART II.

III. Z8 SERVO PROTOCOL

The protocol between the Z8 SERVO microcomputer and the CONTROLLER is based on five I/O lines. Two of the I/O lines are serial input (to Z8 servo from controller) serial output (from Z8 servo to controller). Data stream between the Z8 servo and controller is 8 bit ASCII with no parity bit (the fifth byte of the command string contains check sum byte use for error checking). There are three additional output lines between the Z8 servo used as control lines to the controller. Combining the two serial I/O lines and the three unidirectional port lines generates the bases of the protocol between the Z8 servo and controller. The important operations between the Z8 servo and controller are:

1. Send commands to Z8 servo.

2. Read Z8 servo status.

3. Check validity of all four command bytes.

4. I/O timing signals between the Z8 servo and controller.

5. Z8 servo reset.

Sequencing the Z8 servo controller is an important process following a Power Up (Power On Reset) or if the controller should issue a Z8 Servo Reset at any time. After a Z8 Servo Reset is inhibited, the Z8 I/O ports and internal register are initialized. This takes approximately 75 msec after the Z8 Servo Reset is inhibited. The protocol baud rate is automatically set to 19.2KB and then the system is parked at HOME position and SIO READY is set active. ***IMPORTANT***. If the desired baud rate needs to be increased to 57.6KB; **after a Z8 Servo Reset is the ONLY time this can be done***. Once set to 57.6KB the communication rate remains at 57.6KB until a Z8 Servo Reset occurs. Setting 57.6KB is achieved as follows:

1. Z8 Servo "Power On or Controller" Reset

2. Wait for SIO Ready

3. Send a READ STATUS COMMAND as follows:

   BYTE 1 = $ 00
   BYTE 2 = $ 00
   BYTE 3 = $ 00
   BYTE 4 = $ 87

After the completion of transmitting the bytes, the Z8 Servo Controller changes to 57.6KB and will be waiting for the next transmitted command at 57.6KB.

Before the controller transmits the command byte the controller must pole the SIO READY line from the Z8 servo to determine if it is active (+5 volts). If the line is active then a command can be transmitted to the Z8 servo. The program in the Z8 servo will determine what to do with the command bytes (depending upon the current status of the Z8 servo). After the command (five bytes long) has been transmitted to the Z8 servo, the program in the Z8 servo will determine if the command bytes (first four bytes) are in error by evaluating the check sum byte (fifth byte transmitted). See Charts III and IV for the error handling procedures. After the controller has transmitted the last serial string it must wait 250 usec then test for SERVO ERROR active (+5 volts). If SERVO ERROR is active the command was rejected (check sum error or invalid command). If SERVO ERROR is set active 600 U sec after the command is sent (and not 250 U sec), this was a command reject. The SERVO ERROR must be cleared by a READ STATUS COMMAND or RECAL COMMAND before transmitting another command. See CHART 1 for the timing diagram of the command sequence and I/O protocol.

As long as SIO READY is active the controller can communicate with the Z8 Servo Controller. If SERVO READY is not active the only command that will cause the Widget Servo to set SERVO READY active is a RECAL COMMAND (NORMAL or FORMAT). Read Status will only clear SERVO ERROR, and all other commands will be rejected.

Next, if SERVO READY is active and SERVO ERROR is also active, SERVO ERROR can be cleared by:

1. Any READ STATUS COMMAND.

2. Any RECAL COMMAND.

3. Any other commands will be rejected and maintain SERVO ERROR.

If a SEEK COMMAND is transmitted with both SERVO READY and SERVO ERROR active, the command will be rejected.

It is important to check the status of all three status lines from the Z8 Servo. It is best to avoid sending a SEEK COMMAND with SERVO READY and SERVO ERROR active.

Chart V, parts A-I, illustrate some of the serial communication commands and error conditions that can occur between the controller and Z8 SERVO.

IV. ERROR HANDLING

SERVO ERROR will be generated during the following conditions:

1. During Recal mode (velocity control only) access time-out.If a Recal function exceeds 150 msec then an access timeout occurs.

2. During Seek mode (velocity control only) access time-out. If a Seek function exceeds 150 msec then an access time-out occurs.

3. During Settling mode (following a Recal, Seek, or Offset) if there is excessive On Track pulses (3 crossings), indicating excessive head motion, a Settling error check will occur.

4. During a command transmission if a communication error occurs (check sum error).

5. During a command tansmission if a invalid command is sent.

APPENDIX A:

I. The purpose of the FINE POSITION SERVO is to maintain detent or lock on a given data track. Any misregistrations of the head/arm due to windage, mechanically observed by the optics position signal are corrected by the close loop position servo. Misregistrations at the data head relative to the actual data track on the disk must be corrected by the AUTO OFFSET command. Figure I is a block diagram of the Widget FINE POSITION SERVO. The amount of misregistration at the data track sensed after an AUTO OFFSET command is summed into the servo and the servo is automatically repositioned over the data track.

II. The COARSE POSITION SERVO (SEEK) has the function of moving the data head arbitrarily from a current track to any other arbitrary track location within the total number of track locations between the inner to outer crash stops. When a command is transmitted to the Z8 Servo controller, the Z8 decodes and interprets the command into a servo function. If a SEEK command is sent to the Z8 Servo Controller a direction and number of tracks to move is also sent. The system starts its move to the new track location. When the arm has moved to its new location the Z8 Servo Controller provides control and delay necessary to allow the data head and the FINE POSITION SERVO to come to rest immediately following a SEEK. This insures that motion in FINE POSITION SERVO and data head will be under control when the READ/WRITE channel begins operation. Reliability of the data channel is assured with high margins. Figure I is a block diagram of the Widget COARSE POSITION SERVO.

The differences between the FINE POSITION SERVO and the COARSE POSITION SERVO is handled by the Z8 Servo Controller. The two servos share for the most part the same set of electronics. The Z8 Servo Controller and analog multiplexers switch between the signal paths. In general there are some circuits that are not shared because of their uniqueness for a particular servo.

APPENDIX B:

An important part of the Widget Servo System is the optics signal. The optics signal provides the necessary signals for the fine position servo to position the data head accurately over the data track and to provide the system velocity signal during seek mode. The alignment of the optics signal is described in the following section on "WIDGET OPTICS ALIGNMENT PROCEDURE."

Dan Retzinger
Nov. 9, 1982

WIDGET OPTICS ALIGNMENT PROCEEDURE

INTRODUCTION

The purpose of this note is to describe the procedure for properly adjusting five pots on the widget mother board used to control the amplitude of the optics signal. The five pots are R7, R8, R17, R19 and R35. The optics signal originates at the end of the servo arm and is used in positioning the arm.

EQUIPMENT REQUIRED

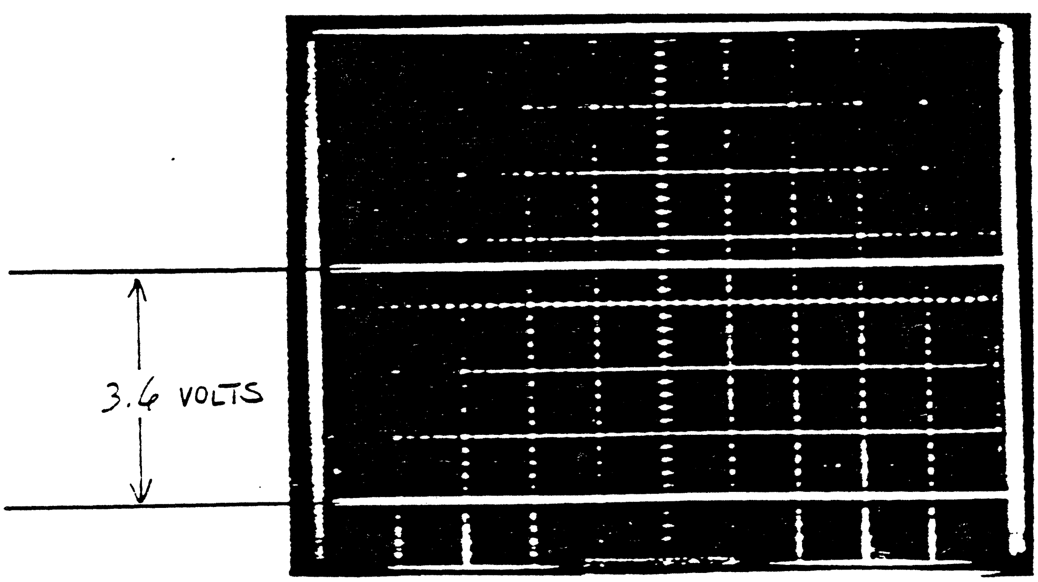
An oscilloscope capable of operating in the X-Y mode of operation. A Tektronix model 465 works fine.

PROCEEDURE

Optics LED Drive Adjustment

1. Connect channel 1 of the oscilloscope to TP 5 on the Widget Mother Board.
2. Scope Vert. setting: 1 Volt/Div. Horizontal: Any sweep rate.
3. Adjust R35 so the voltage at TP5 is 3.6 volts +/- .2 volts. 1.9v - 3.8v
   (clockwise, or more resistance=lower voltage)

Figure 1: TP5 Amplitude



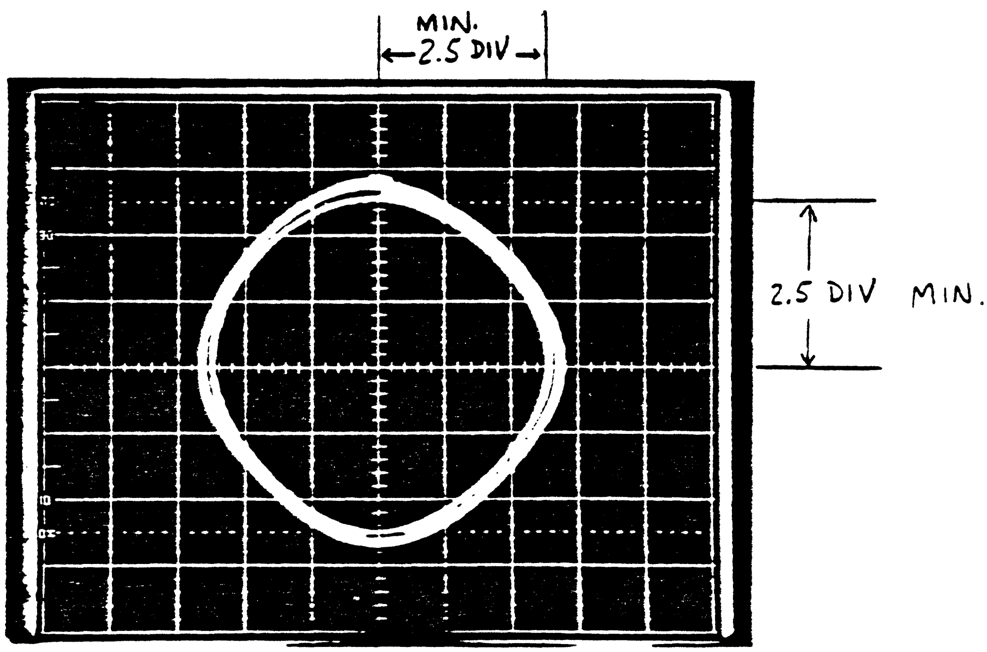
Position A and Position B Adjustment

4. Put scope in X-Y mode, ground channels X and Y, move dot to center of screen.
5. Connect chan X to TP9, chan Y to TP8. (Both TP's are located near pin 1 of the Z8 microprocessor)
6. Scope vertical: Chan X and Y, 2 volts/Div.
7. At this point arm is to be moved. ** to be determined how **
8. With arm in movement, a circular pattern should appear on the scope. Adjust R7, R8, R17, R19 so the top, bottom, right and left sides of the circle come at but no closer than a minimum of 2.5 scope divisions from the center of the screen.
9. Each pot adjusts the circle as follows:

R7	Left side	clockwise or lower res=smaller circle
R8	Right side	"
R17	Bottom	"
R19	Top	"

10. Figure 2 shows a properly adjusted optics signal.

Figure 2: Position A and B



PROCEEDURE SUMMARY

1. Adjust R35 so the voltage at TP5 (R37) is 3.6 Volts +/- .2 volts.

2. Put scope in X-Y mode, chan 1 & 2 set to 2 volts/div. Adjust R7, R8, R17, R19, so that the sides of the circle (during minimum fluctuation) are each within 2.5 Divisions (+/- .1 div) of the center. This corresponds to 5 Volts from the center to the top, bottom, or either side.

APPENDIX C:

Some of the analog control signals can be useful in understanding or evaluating the function or performance of the Widget Servo. Photographs are provided to illustrate some of the key Widget functions. Refer to the following document "WIDGET SERVO WAVEFORMS."

WIDGET SERVO

VARIOUS KEY WAVEFORMS

CONTENTS

WAVEFORM: Optics Adjustment

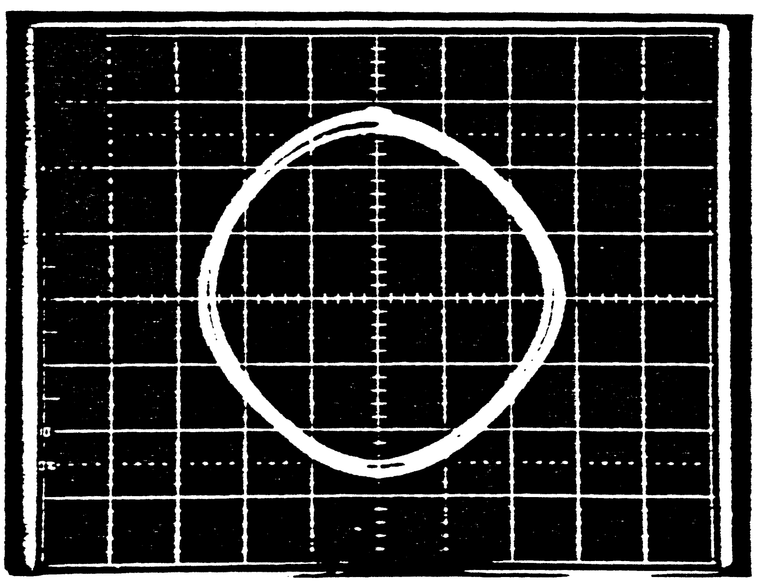
Scope Adjustments:

Channel	Probe Tip	Test Point	Notes
Chan 1	Position A	TP9	2V/div
Chan 2	Position B	TP8	2V/div
Trig In	Not used		
Horiz :	X-Y Mode		

Servo:

Alternate Seeks, 512 tracks

Press Z; 82, 0, 0, 0
86, 0, 0, 0



WAVEFORM: Current Sense and Position A

Scope Adjustments:

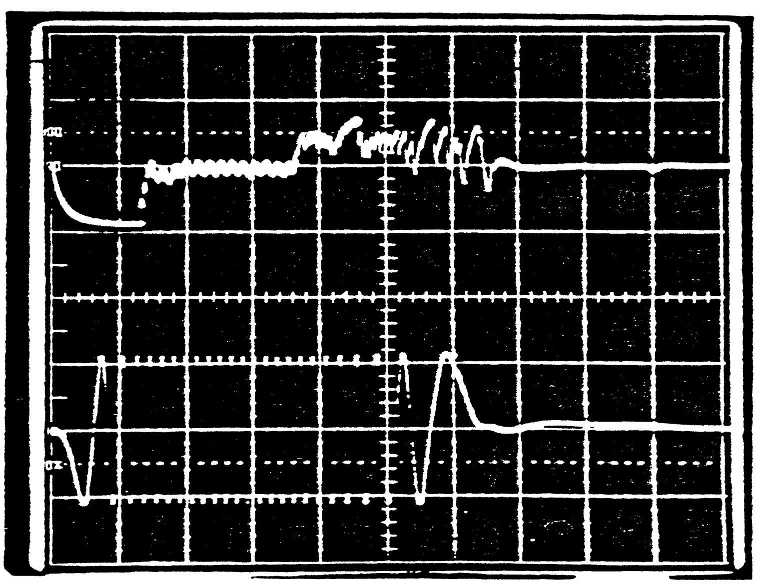
Channel	Probe Tip	Test Point	Notes
Chan 1	Current Sense	TP19	5V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 5ms/Div Calibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: Current Sense and Position A
(Forward and Reverse Seeks)

Scope Adjustments:

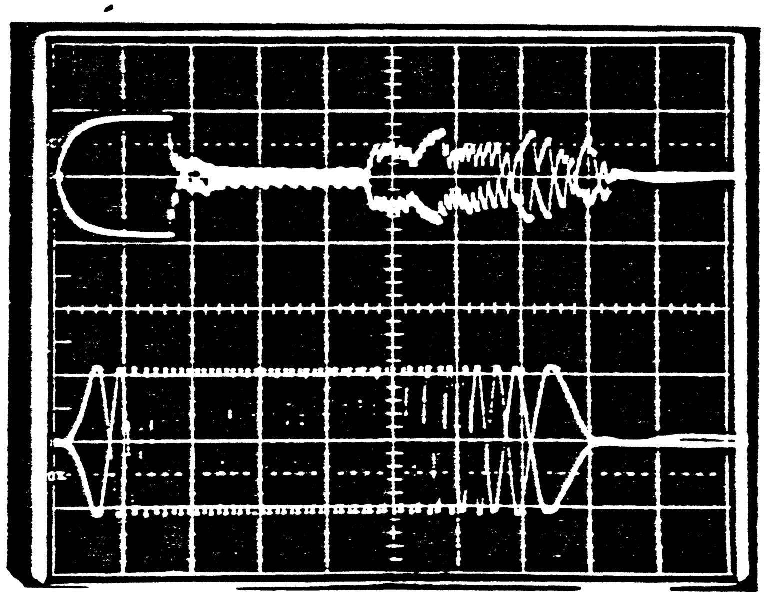
Channel	Probe Tip	Test Point	Notes
Chan 1	Current Sense	TP19	5V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 2ms/Div Uncalibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: Velocity and Position A

Scope Adjustments:

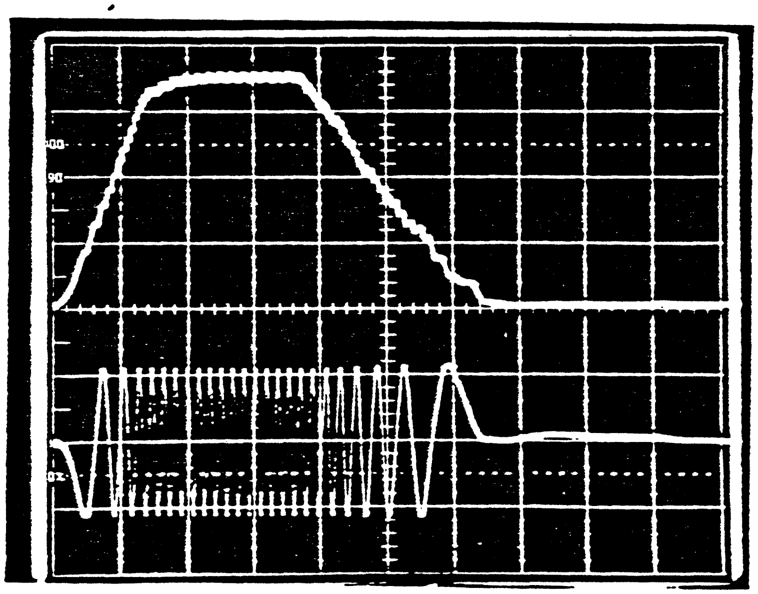
Channel	Probe Tip	Test Point	Notes
Chan 1	Velocity	TP7	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 5ms/Div Calibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: Velocity and Position A
(Forward and Rev Seeks)

Scope Adjustments:

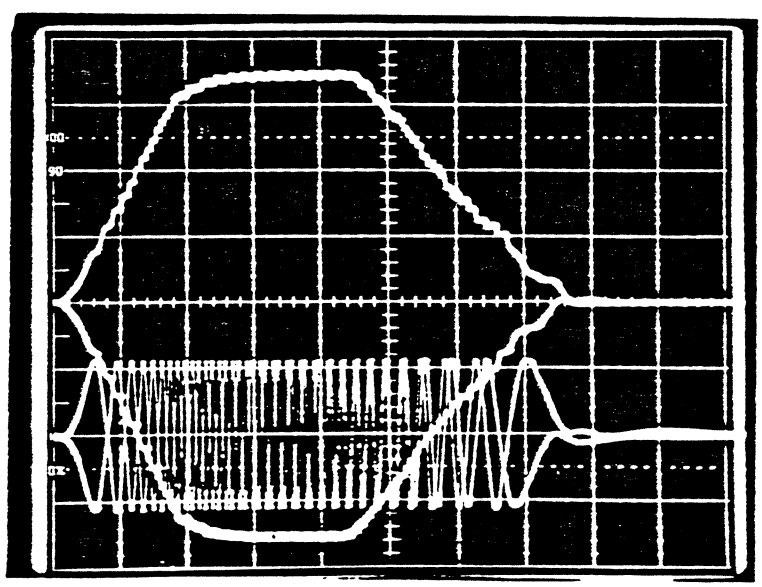
Channel	Probe Tip	Test Point	Notes
Chan 1	Velocity	TP7	5V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 2ms/Div Uncalibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: DAC Output and Position A

Scope Adjustments:

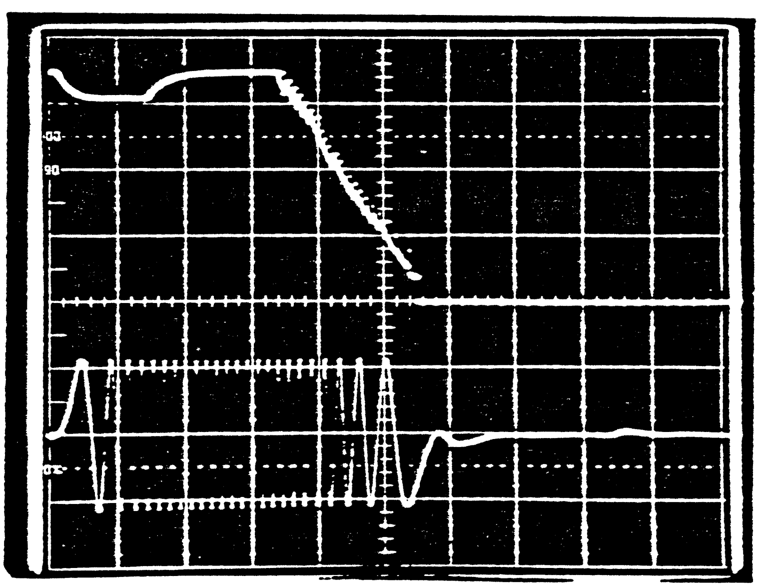
Channel	Probe Tip	Test Point	Notes
Chan 1	DAC Output	TP13	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 5ms/Div Calibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: DAC Output and Position A
(Forward and Rev Seeks)

Scope Adjustments:

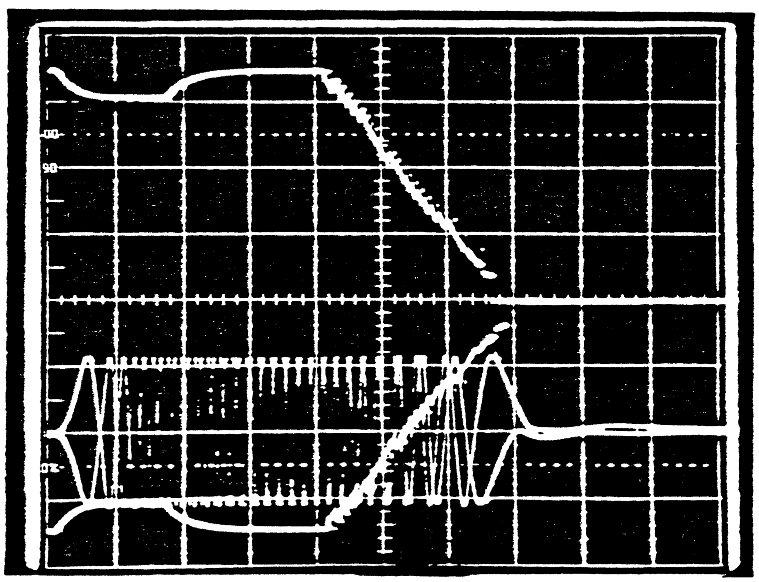
Channel	Probe Tip	Test Point	Notes
Chan 1	DAC Output	TP13	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 2ms/Div Uncalibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



WAVEFORM: Curve Shift Function and Position A
(Forward and Rev Seeks: 1 track)

Scope Adjustments:

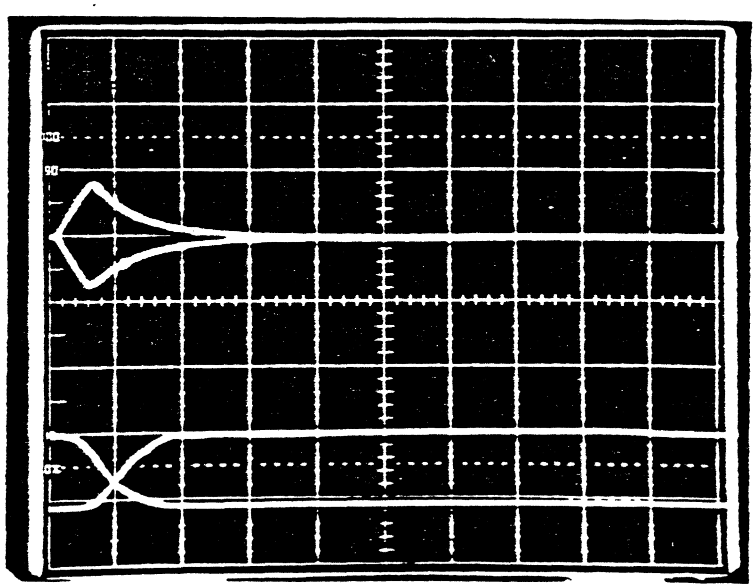
Channel	Probe Tip	Test Point	Notes
Chan 1	Curve Shift Func.	TP12	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 2ms/Div Uncalibrated

Servo:

Alternate Seeks, 1 track

Press Z; 80, 01, 0, 0
84, 01, 0, 0



WAVEFORM: Curve Shift Function and Position A
(60 track seek)

Scope Adjustments:

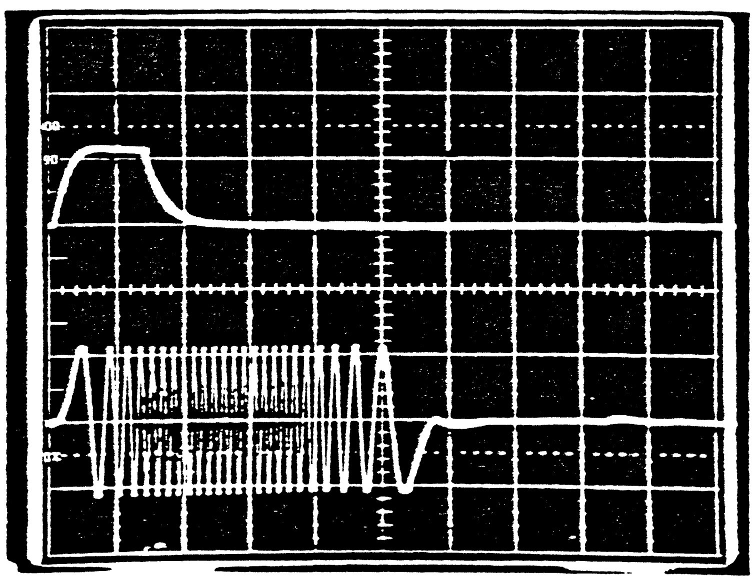
Channel	Probe Tip	Test Point	Notes
Chan 1	Curve Shift Func.	TP12	2V/div
Chan 2	Position A	TP9	5V/div
Trig In	Access Mode	TP27	Positive trig, Ext/10

Horiz: 5ms/Div Calibrated

Servo:

Alternate Seeks, 96 tracks (Hex $60)

Press Z; 80, 60, 0, 0
84, 60, 0, 0



I. BYTE 1: COMMAND BYTE (DIFCNTH)

```
          ---
          !B7
command   !B6
bits      !B5
          !B4
          ---
          ---
          !B3 -X- not used
access    !B2 -access direction
bits      !B1 -hi diff2 (512)
          !B0 -hi diff1 (256)
          ---
```

B7	B6	B5	B4	FUNCTIONS
1	0	0	0	access only
1	0	0	1	access with offset
0	1	0	0	normal recal (to trk 72)
0	1	1	1	format recal (to trk 32)
0	0	0	1	offset-trk following
1	1	0	0	home-send to ID stop
0	0	1	0	diagnostic command
0	0	0	0	read status command

```
access direction = 1 (FORWARD: toward the spindle)
                 = 0 (REVERSE: away from the spindle)

hi diff2 (512)   = 1 (512 tracks to go)
                 = 0 (not set)

hi diff1 (256)   = 1 (256 tracks to go)
                 = 0 (not set)
```

II. BYTE 2: DIFF BYTE (DIFCNTL)

command BYTE 2 contains the LOW ORDER DIFFERENCE COUNT for a seek

```
 ---
!B7 -bit7= 128 tracks
!B6 -bit6= 64  tracks
!B5 -bit5= 32  tracks
!B4 -bit4= 16  tracks
!B3 -bit3= 8   tracks
!B2 -bit2= 4   tracks
!B1 -bit1= 2   tracks
!B0 -bit0= 1   track
 ---
```

III. BYTE 3: OFFSET BYTE (STATREG)

command BYTE 3 contains the INSTRUCTION for an OFFSET COMMAND (seek or during track following)

```
 ---
|B7 -offset direction
|B6 -auto offset function
|B5 -read offset value (after auto or manual)  'NOT USED'
|B4 -offset bit4 =16
|B3 -offset bit3 =8
|B2 -offset bit2 =4
|B1 -offset bit1 =2
|B0 -offset bit0 =1
 ---
```

1. if offset command from BYTE 1 is followed by bit6 set (auto offset offset direction (bit7) read offset (bit5) and bits 4-0 are ignore but should be set to 0 if not used.

2. OFFSET DIRECTION =1 (FORWARD OFFSET:toward the spindle)
   =0 (REVERSE OFFSET:away from the spindle)

3. AUTO OFFSET =1 (normally used preceeding a write operation)
   =0 (manual offset:MUST send direction and magnitu of offset)

~~4. READ OFFSET =1 (read offset value from DAC; i.e. after auto offset)~~
   ~~=0 (no action)~~

~~* READ OFFSET COMMAND desired after AUTO OFFSET MUST be sent as two seperate commands~~

IV. BYTE 4: STATUS BYTE (CNTREG)

```
 ---
|B7 -communication rate
|B6 -power on reset
|B5 -not used
|B4 -not used
|B3 -status or diagnostic bits
|B2 -              |
|B1 -              |
|B0 -              V
 ---
```

B7=0; Communication Rate is 19.2 KBAUD
  =1; Communication Rate is 57.6 KBAUD

B6=0; Power On Reset bit is no active
  =1; Power On Reset bit is active

. BYTE 5: CHECKSUM BYTE (CKSUM)

[B7 B6 B5 B4 B3 B2 B1 B0]

results of the transmitted CHECKSUM BYTE are derived as:

$\overline{\text{(BYTE 1 + BYTE 2 + BYTE 3 + BYTE 4)}}$ = CHECKSUM BYTE

(+) is defined as the addition of each BYTE

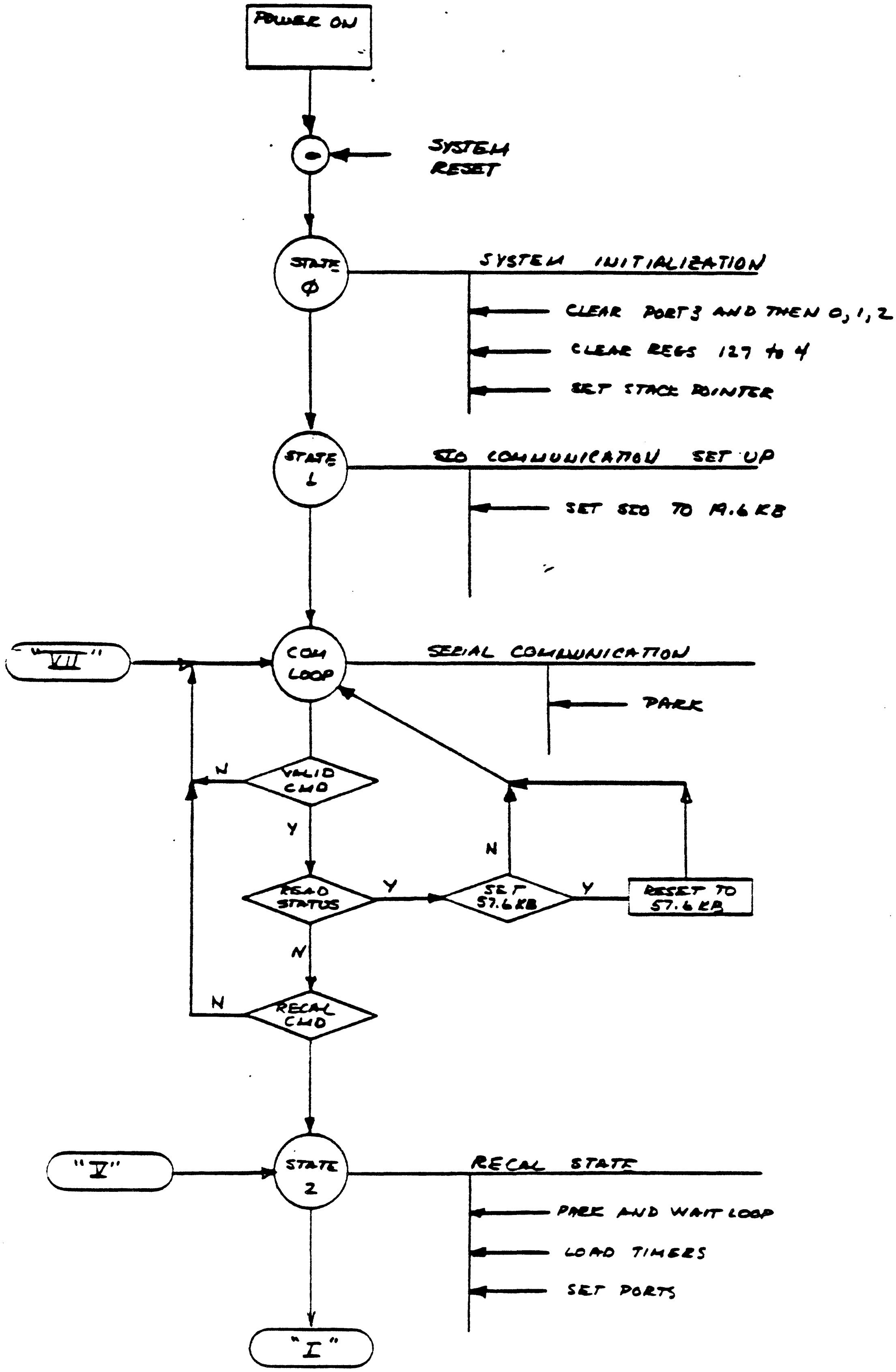
($\overline{\text{BYTE}}$) is defined as the compliment of the BYTES(1-4)

VI. The SERVO STATUS lines (SIO RDY,SERVO RDY,SERVO ERROR) must have the following conditions in order to send the listed Z8 COMMANDS:

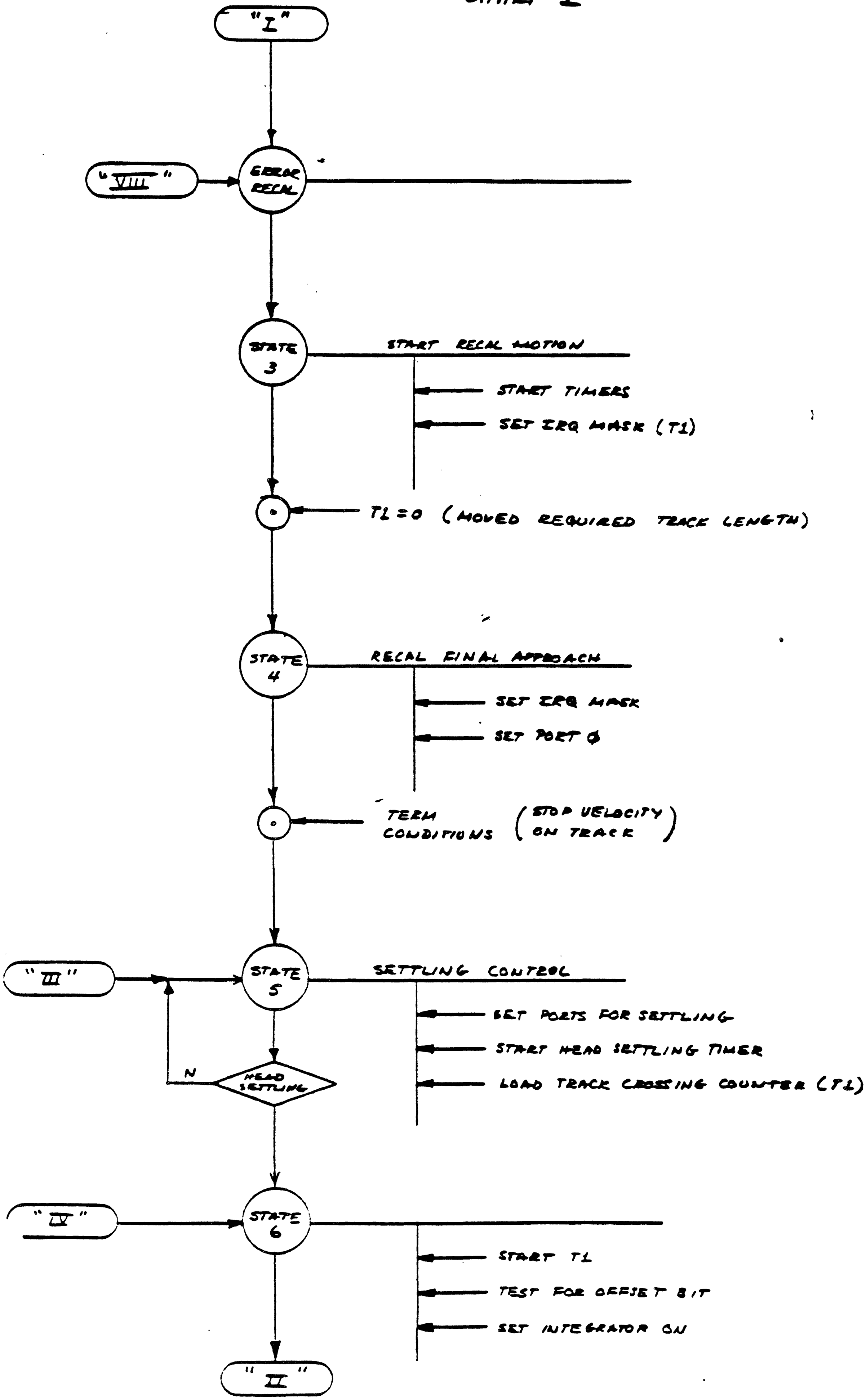
Z8 SERVO CMD	HEX	SERVO STATUS: SIO RDY	SRV RDY	SRV ERR	
access(only)	8X	1	1	0	
access(offset)	9X	1	1	0	
recal(data)	40	1	X	X	
recal(format)	70	1	X	X	
park	C0	1	X	X	
offset(detent)	10	1	1	0	
status	00	1	X	X	
diagnostic	20	------------			not implimented

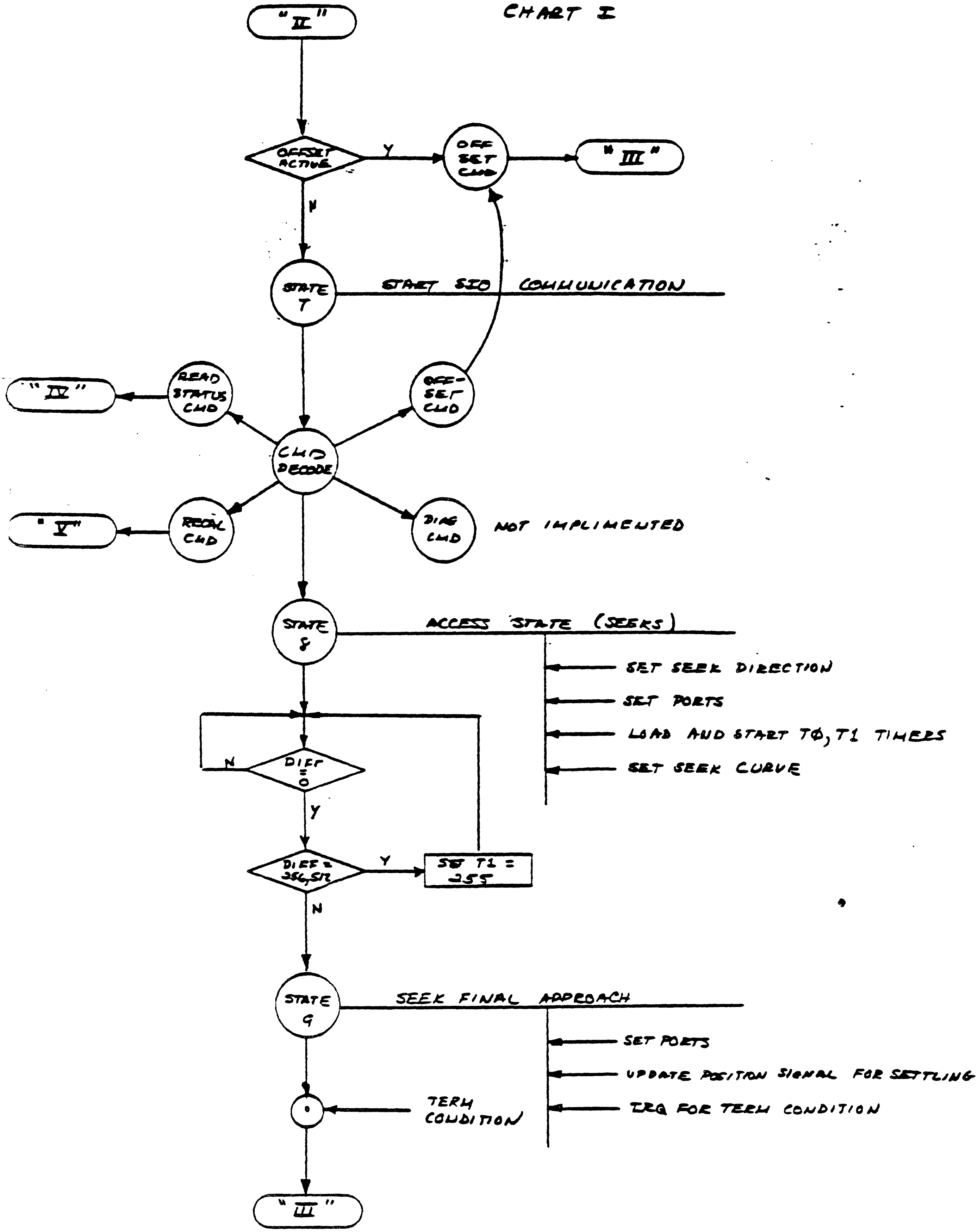
X= either 0,1

Z8 SERVO SEQUENCER
CHART I



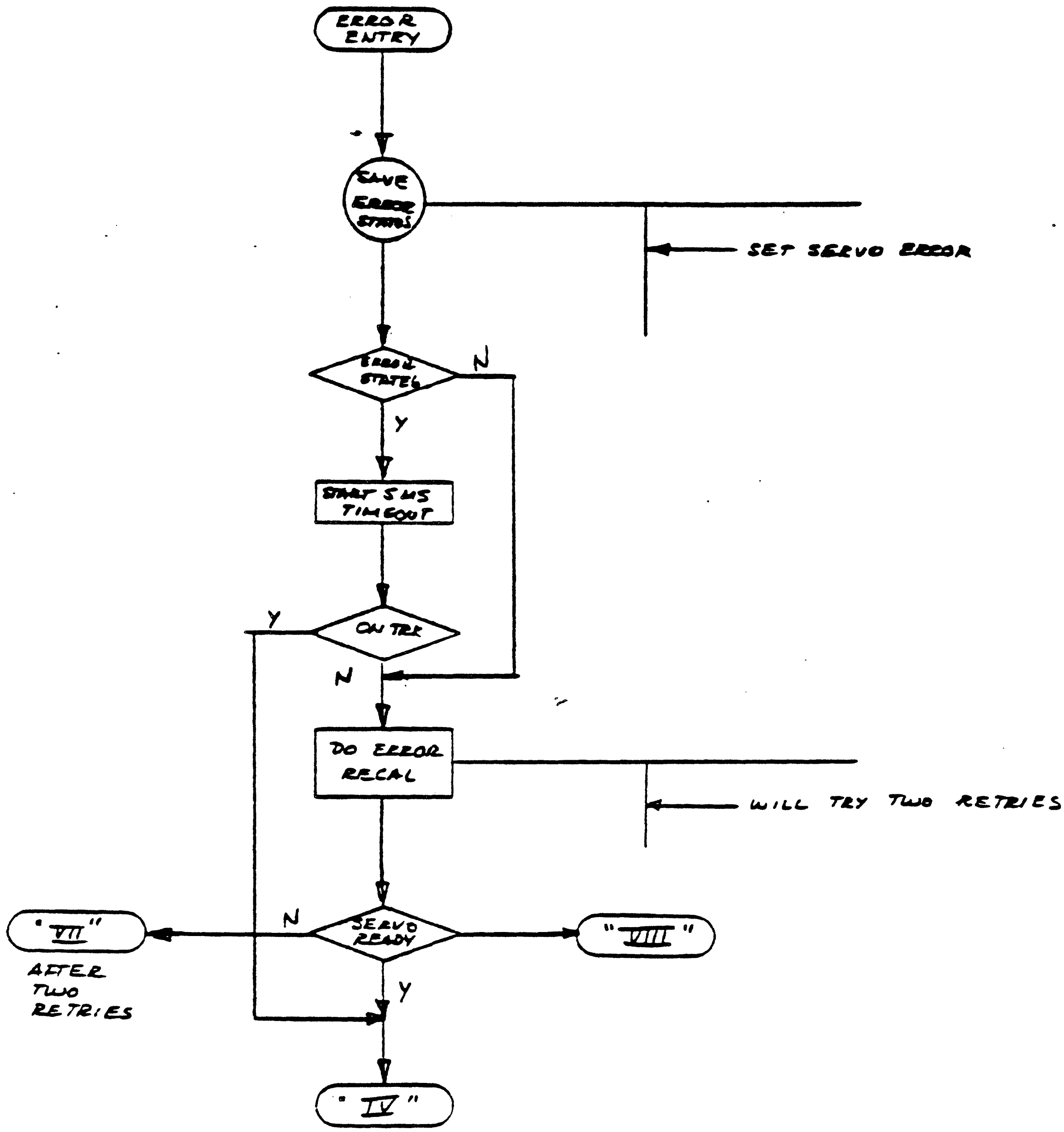
Z8 SERVO SEQUENCER
CHART I



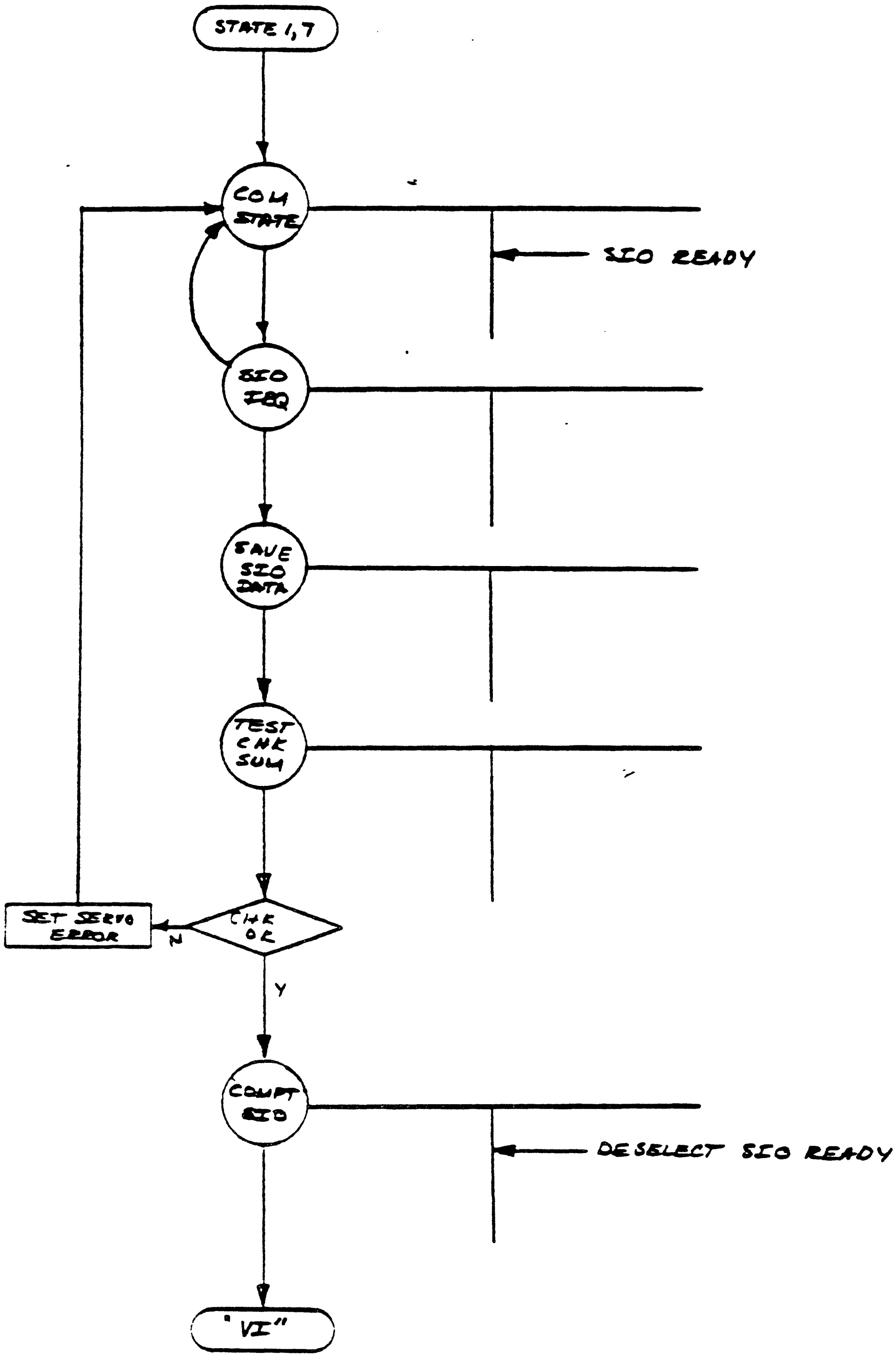

"II"
OFFSET ACTIVE
Y
N
OFF SET CMD
"III"
STATE 7
START SIO COMMUNICATION
"IV"
READ STATUS CMD
OFF-SET CMD
CMD DECODE
"V"
RECAL CMD
DIAG CMD
NOT IMPLIMENTED
STATE 8
ACCESS STATE (SEEKS)
SET SEEK DIRECTION
SET PORTS
LOAD AND START TØ, T1 TIMERS
SET SEEK CURVE
DIFF = 0
DIFF = 256,512
SET T1 = 255
STATE 9
SEEK FINAL APPROACH
SET PORTS
UPDATE POSITION SIGNAL FOR SETTLING
IRQ FOR TERM CONDITION
TERM CONDITION
"III"

SERVO ERROR

CHART II



COMMUNICATION ERRORS
CHART III



COMMAND ERRORS

CHART IV

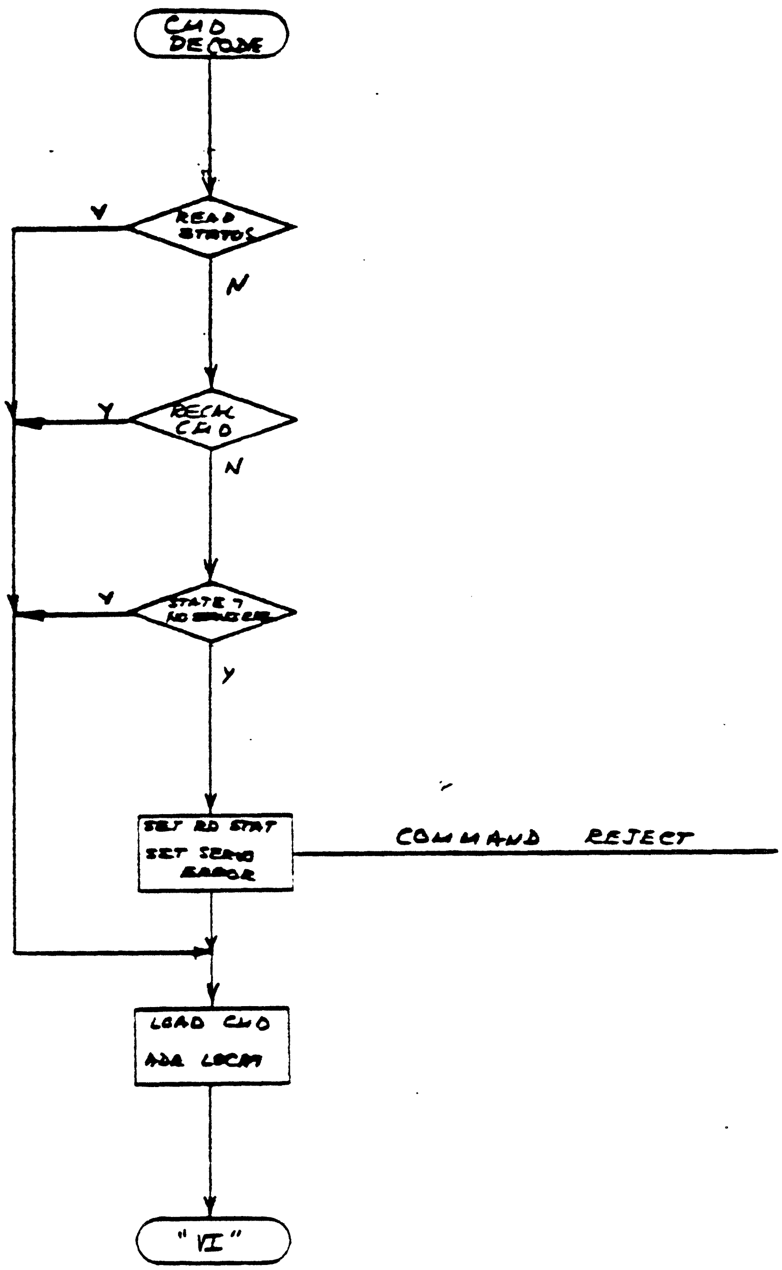
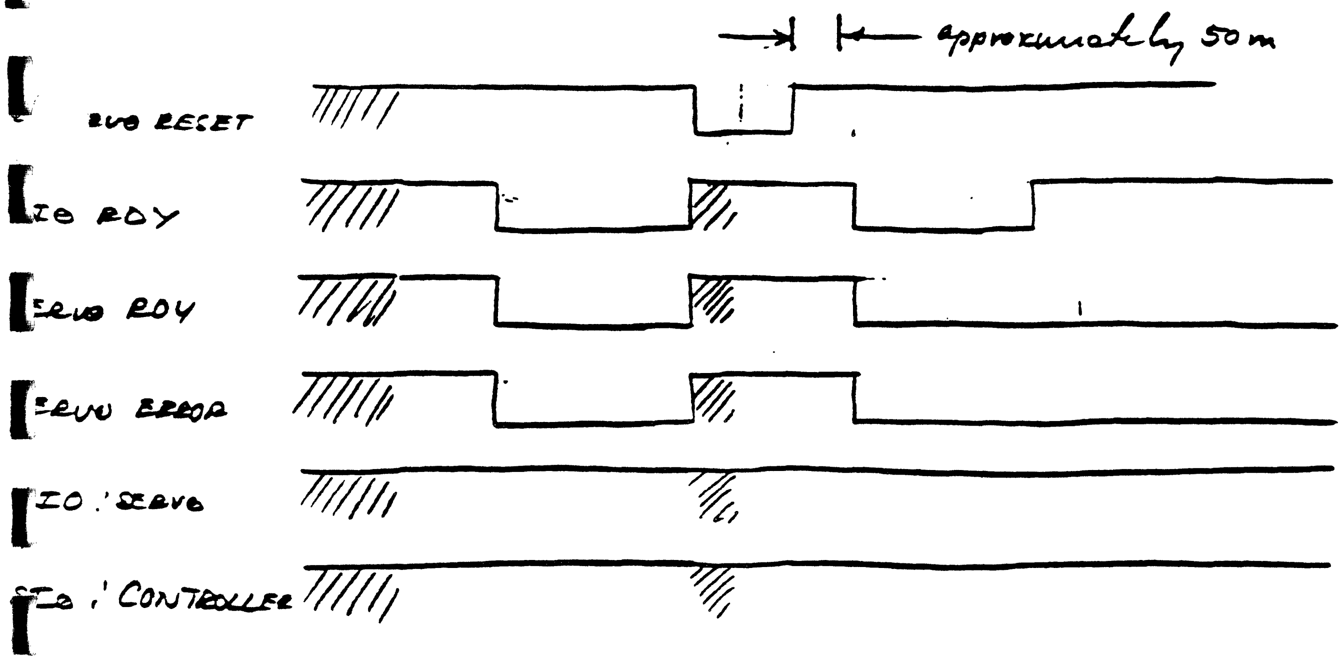
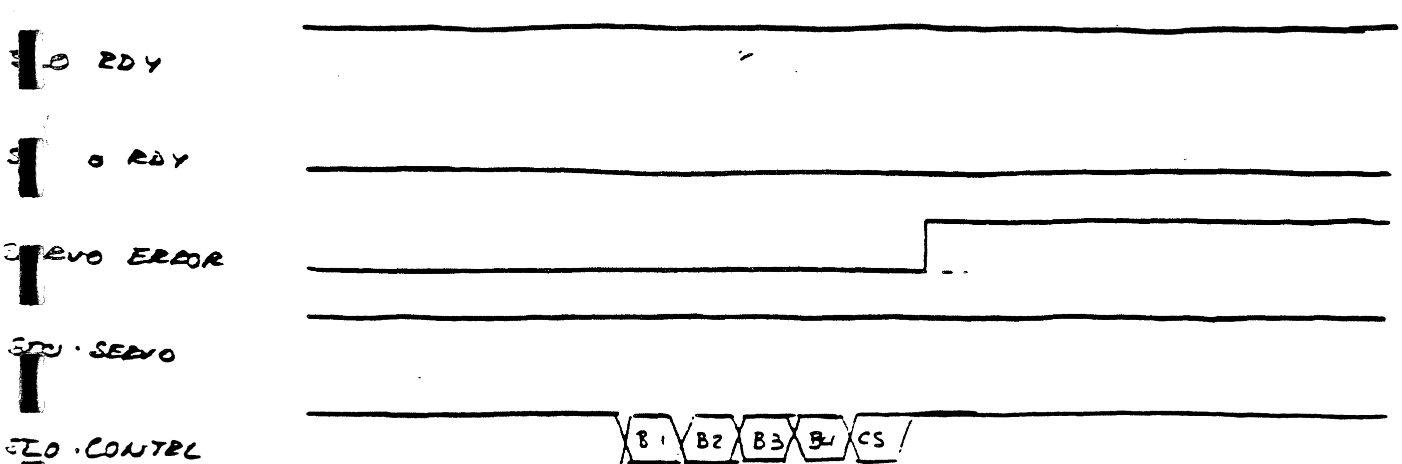


CHART V

A - POWER UP



B - AFTER POWER UP - CHECK SUM ERROR



C - AFTER POWER UP - INVALID CMD

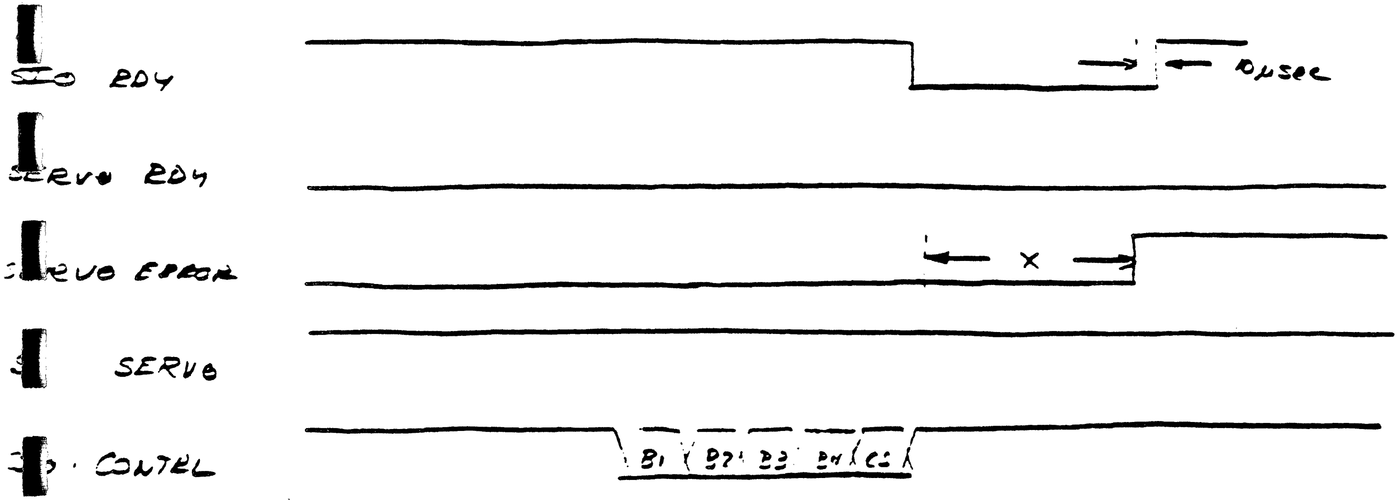
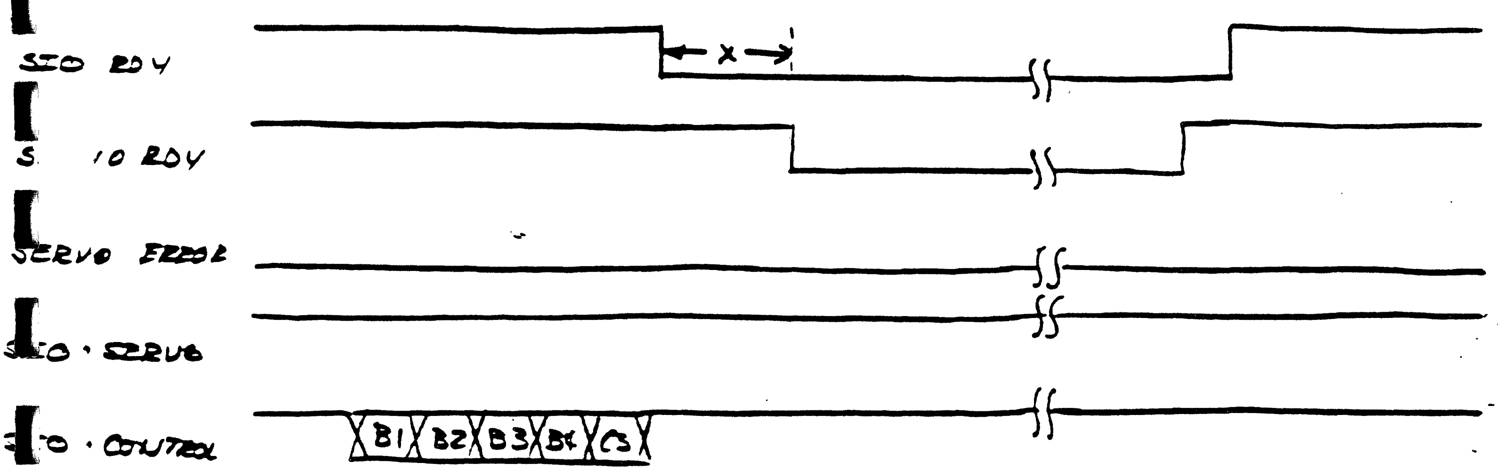
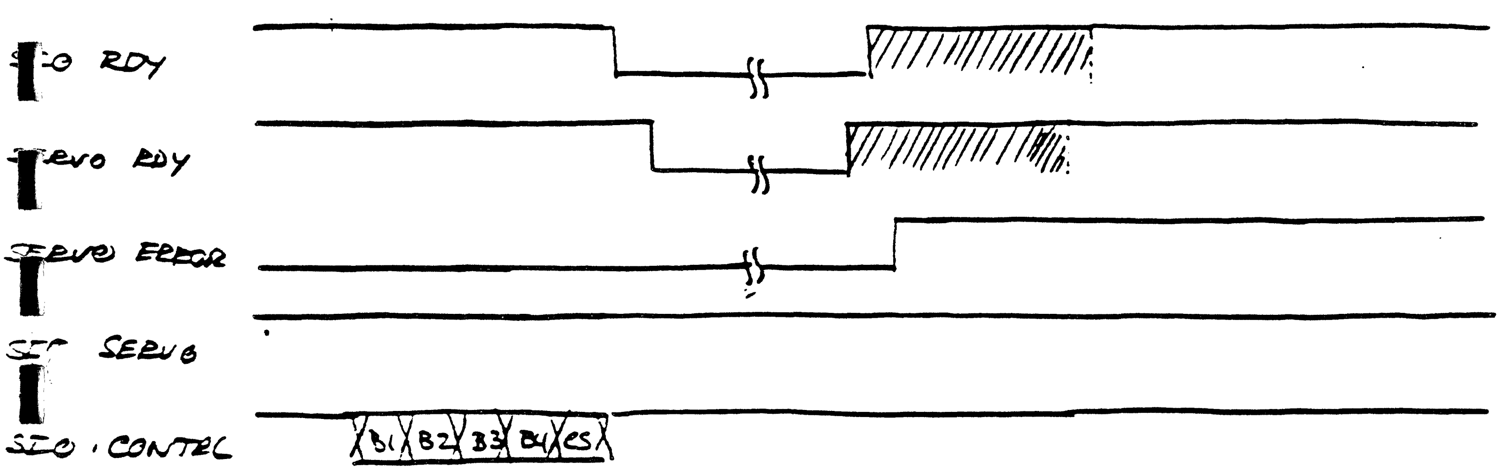


CHART V

G—TRACK FOLLOWING VALID COMMAND (MOVE)



H—TRACK FOLLOWING (MOVE CMD) FOLLOWED BY SERVO ERROR



I—TRACK FOLLOWING (NO COMMAND) SERVO ERROR

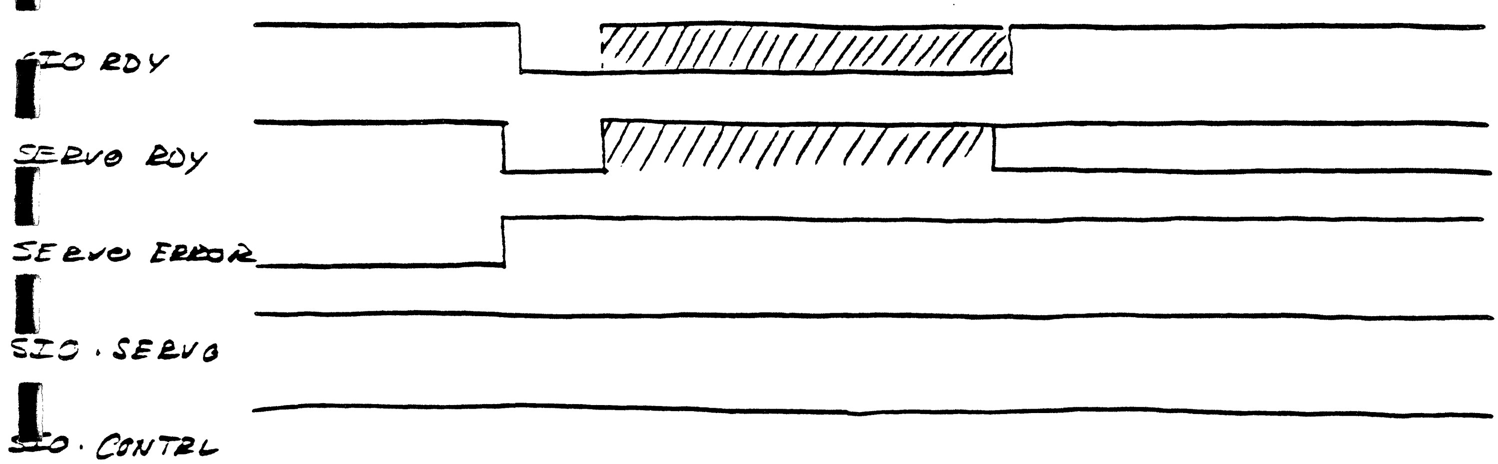
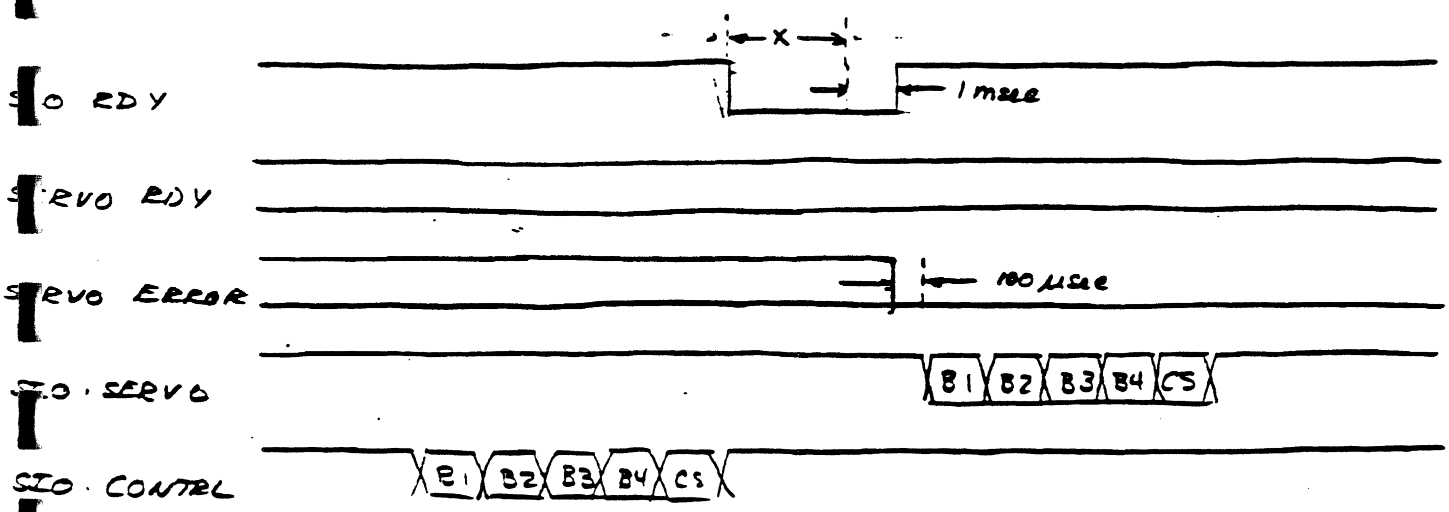
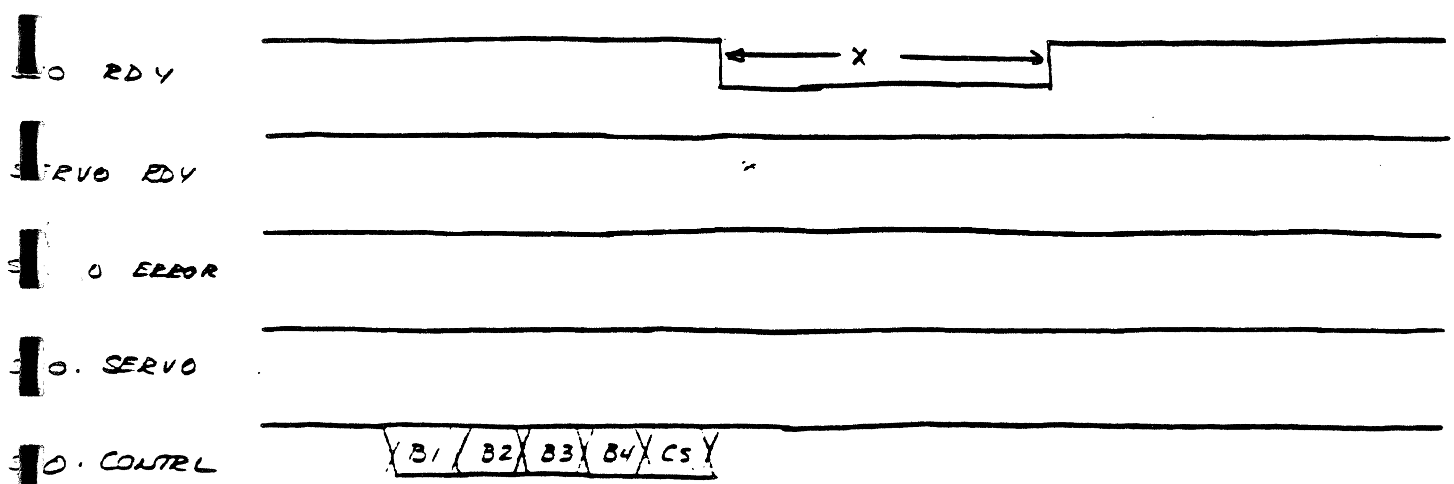


CHART V

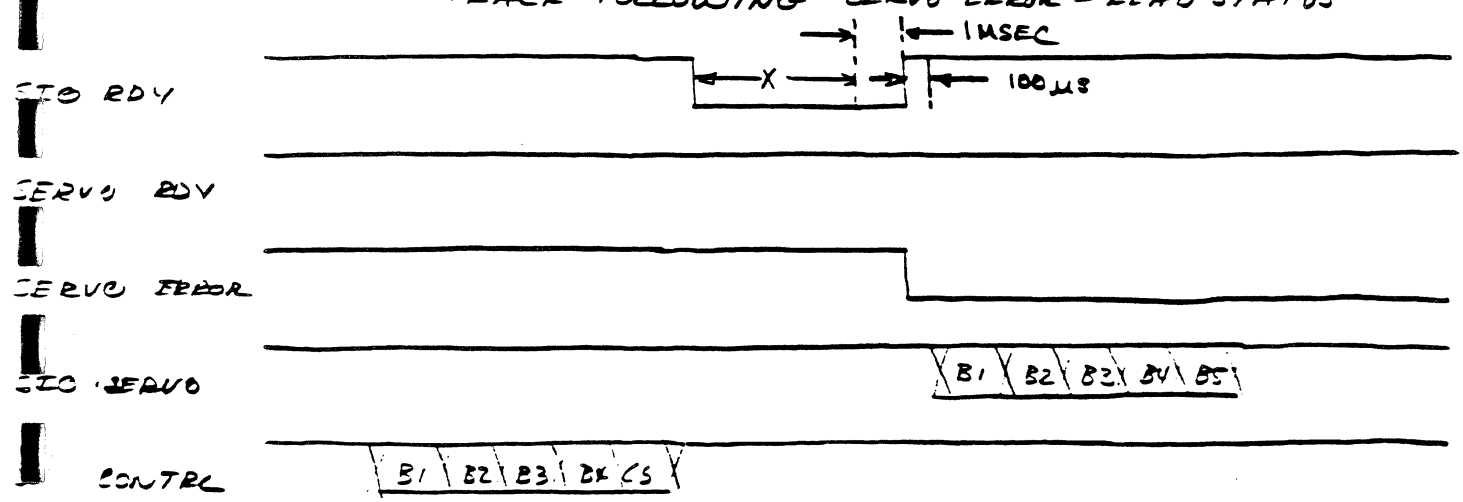
D – READ STATUS COMMAND



E – TRACK FOLLOWING SERVO ERROR – INVALID COMMAND



F – TRACK FOLLOWING SERVO ERROR – READ STATUS



[redacted]

(FINE AND COURSE POSITION SERVOS)

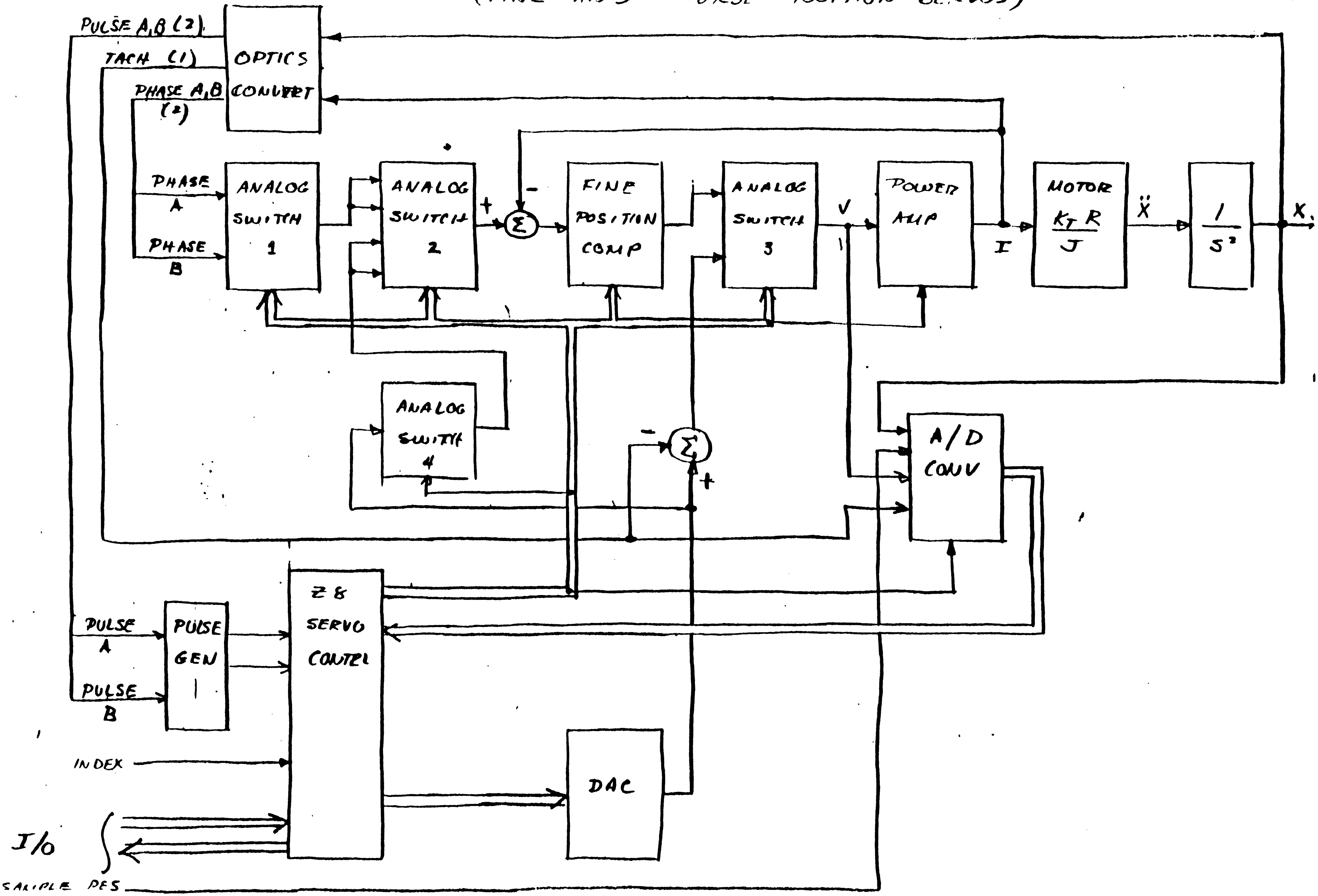
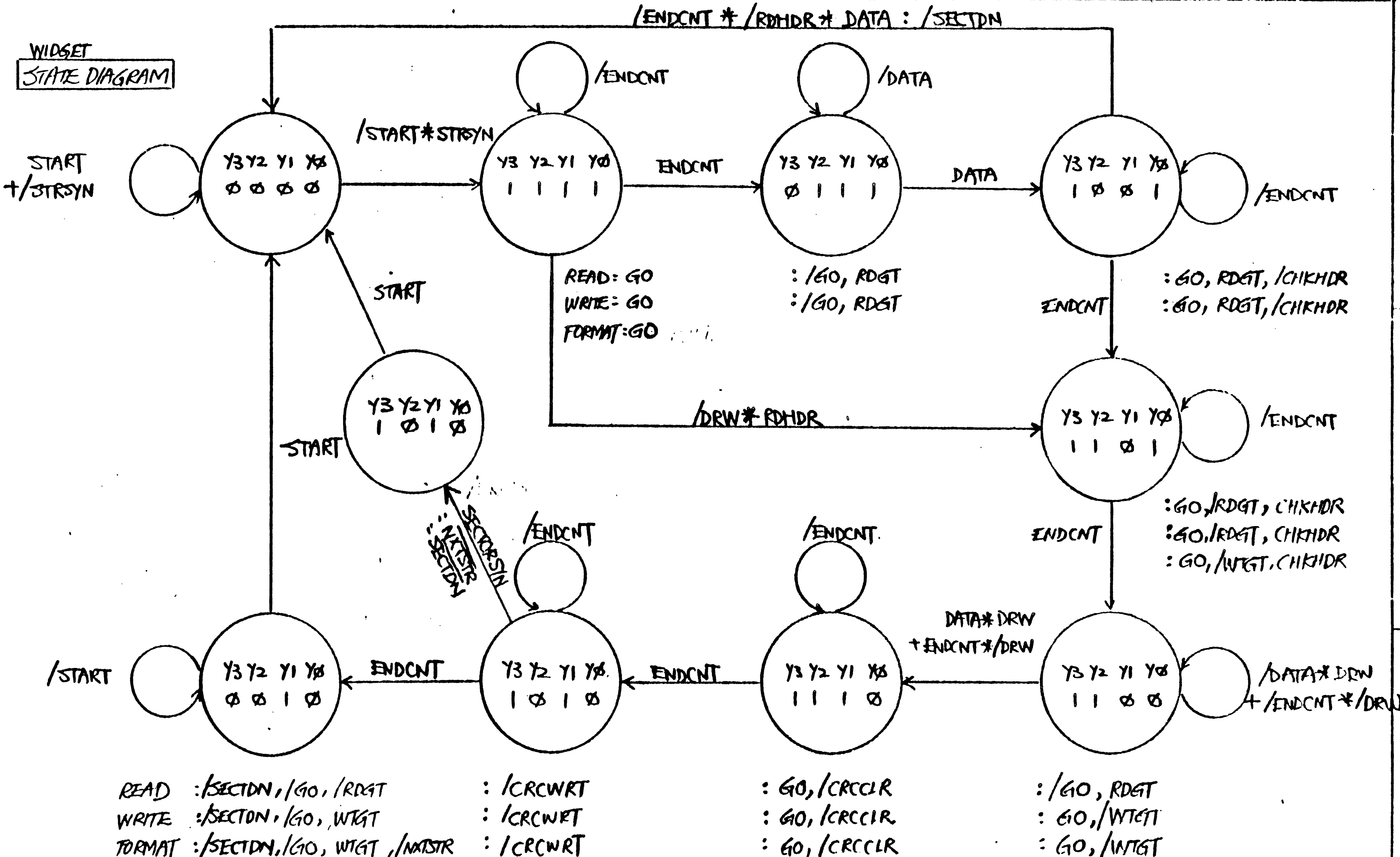


FIGURE I

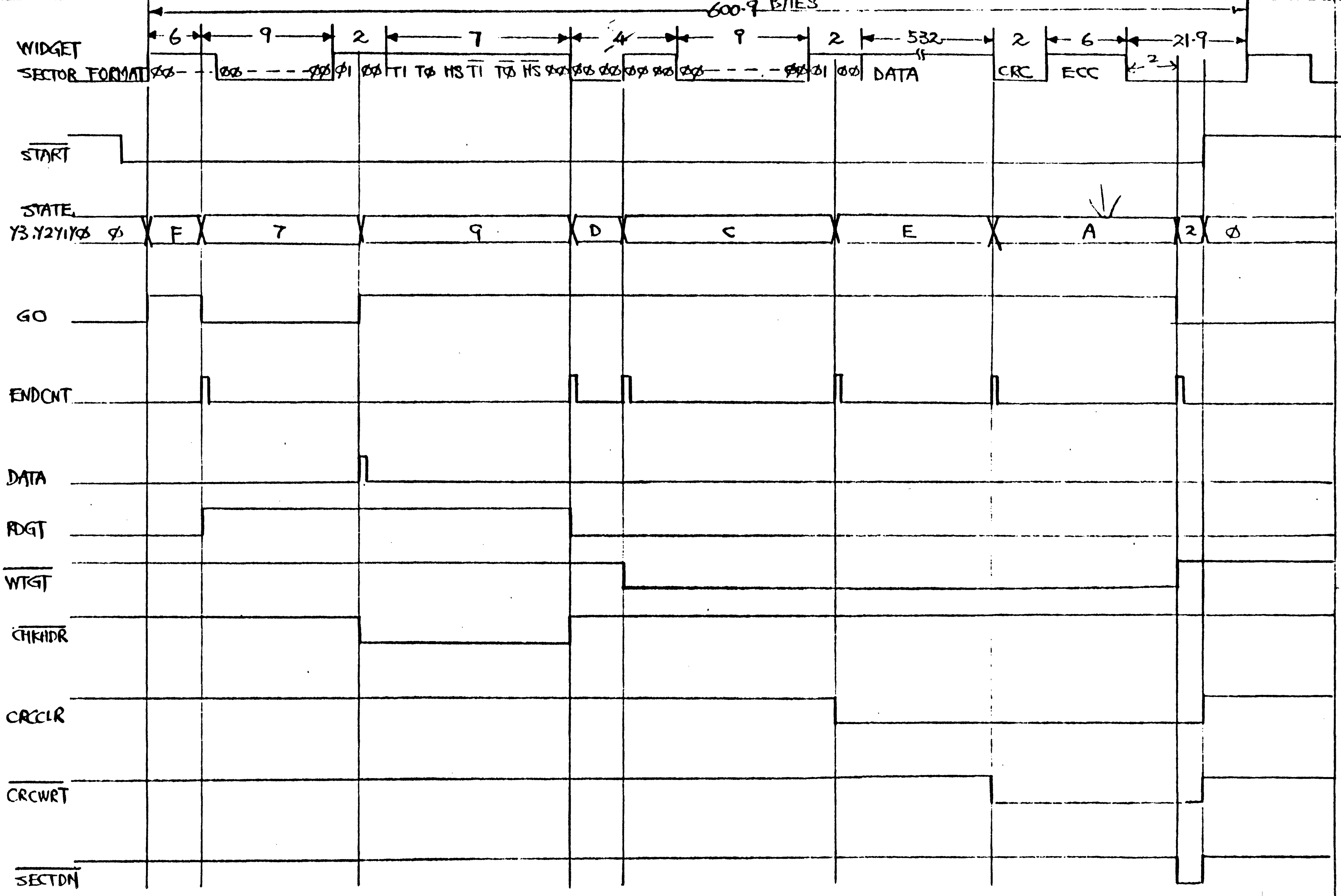
10/18/82


WIDGET
STATE DIAGRAM
/ENDCNT * /RDHDR * DATA : /SECTDN
START
+/STRSYN
Y3 Y2 Y1 YØ
Ø Ø Ø Ø
/START*STRSYN
/ENDCNT
Y3 Y2 Y1 YØ
1 1 1 1
ENDCNT
/DATA
Y3 Y2 Y1 YØ
Ø 1 1 1
DATA
Y3 Y2 Y1 YØ
1 Ø Ø 1
/ENDCNT
READ: GO
WRITE: GO
FORMAT: GO
: /GO, RDGT
: /GO, RDGT
: GO, RDGT, /CHKHDR
: GO, RDGT, /CHKHDR
ENDCNT
START
Y3 Y2 Y1 YØ
1 Ø 1 Ø
START
/DRW* RDHDR
Y3 Y2 Y1 YØ
1 1 Ø 1
/ENDCNT
: GO, /RDGT, CHKHDR
: GO, /RDGT, CHKHDR
: GO, /WTGT, CHKHDR
ENDCNT
SECTDN
: NXTSTR
SECTDN
/ENDCNT
/ENDCNT
DATA* DRW
+ENDCNT*/DRW
/START
Y3 Y2 Y1 YØ
Ø Ø 1 Ø
ENDCNT
Y3 Y2 Y1 YØ
1 Ø 1 Ø
ENDCNT
Y3 Y2 Y1 YØ
1 1 1 Ø
Y3 Y2 Y1 YØ
1 1 Ø Ø
/DATA* DRW
+ /ENDCNT */DRW
READ :/SECTDN, /GO, /RDGT
WRITE :/SECTDN, /GO, WTGT
FORMAT :/SECTDN, /GO, WTGT, /NXTSTR
: /CRCWRT
: /CRCWRT
: /CRCWRT
: GO, /CRCCLR
: GO, /CRCCLR
: GO, /CRCCLR
: /GO, RDGT
: GO, /WTGT
: GO, /WTGT

10/18/82

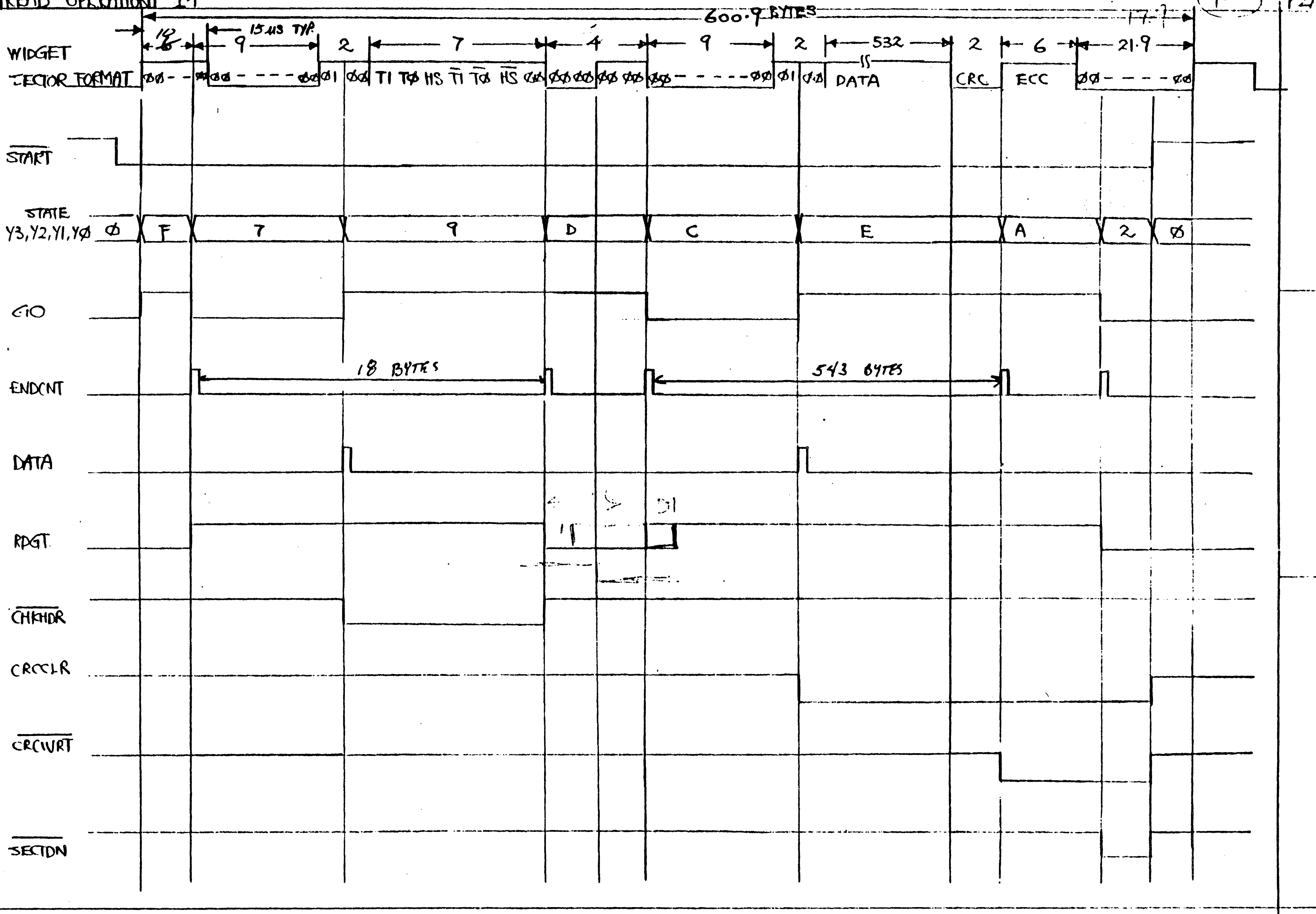
WRITE OPERATION

P3A



10/18/82

READ OPERATION



Mac MFS Boot - Block 810